ConfigurationDesk

Custom I/O Function Implementation Guide

For ConfigurationDesk 6.7

Release 2021-A – May 2021

dSPACE

How to Contact dSPACE

Mail:	dSPACE GmbH Rathenaustraße 26 33102 Paderborn Germany
Tel.:	+49 5251 1638-0
Fax:	+49 5251 16198-0
E-mail:	info@dspace.de
Web:	http://www.dspace.com

How to Contact dSPACE Support

If you encounter a problem when using dSPACE products, contact your local dSPACE representative:

- Local dSPACE companies and distributors: http://www.dspace.com/go/locations
- For countries not listed, contact dSPACE GmbH in Paderborn, Germany.
  Tel.: +49 5251 1638-941 or e-mail: support@dspace.de

You can also use the support request form: http://www.dspace.com/go/supportrequest. If you are logged on to mydSPACE, you are automatically identified and do not need to add your contact details manually.

If possible, always provide the relevant dSPACE License ID or the serial number of the CmContainer in your support request.

Software Updates and Patches

dSPACE strongly recommends that you download and install the most recent patches for your current dSPACE installation. Visit http://www.dspace.com/go/patches for software updates and patches.

Important Notice

Contents

About This Document

Content

This document shows you how to create custom function blocks. Custom function blocks are a way to include user-specific I/O functionality in ConfigurationDesk.

Required knowledge

You should be familiar with:

- The structure and elements of standard function blocks in ConfigurationDesk
- The C++ programming language and the XML language
- If you want to use I/O events, you have to be familiar with modeling applications and tasks. For more information, refer to Introduction to Modeling Executable Applications and Tasks (ConfigurationDesk Real-Time Implementation Guide 🕮).

Symbols

dSPACE user documentation uses the following symbols:

Symbol	Description
⚠ DANGER	Indicates a hazardous situation that, if not avoided, will result in death or serious injury.
⚠ WARNING	Indicates a hazardous situation that, if not avoided, could result in death or serious injury.
⚠ CAUTION	Indicates a hazardous situation that, if not avoided, could result in minor or moderate injury.
NOTICE	Indicates a hazard that, if not avoided, could result in property damage.
Note	Indicates important information that you should take into account to avoid malfunctions.
Tip	Indicates tips that can make your work easier.
⍰	Indicates a link that refers to a definition in the glossary, which you can find at the end of the document unless stated otherwise.

Symbol	Description
📖	Precedes the document title in a link that refers to another document.

Naming conventions

dSPACE user documentation uses the following naming conventions:

%name% Names enclosed in percent signs refer to environment variables for file and path names.

< > Angle brackets contain wildcard characters or placeholders for variable file and path names, etc.

Special folders

Some software products use the following special folders:

Common Program Data folder A standard folder for application-specific configuration data that is used by all users.

`%PROGRAMDATA%\dSPACE\<InstallationGUID>\<ProductName>`

or

`%PROGRAMDATA%\dSPACE\<ProductName>\<VersionNumber>`

Documents folder A standard folder for user-specific documents.

`%USERPROFILE%\Documents\dSPACE\<ProductName>\<VersionNumber>`

Local Program Data folder A standard folder for application-specific configuration data that is used by the current, non-roaming user.

`%USERPROFILE%\AppData\Local\dSPACE\<InstallationGUID>\<ProductName>`

Accessing dSPACE Help and PDF Files

After you install and decrypt dSPACE software, the documentation for the installed products is available in dSPACE Help and as PDF files.

dSPACE Help (local) You can open your local installation of dSPACE Help:

- On its home page via Windows Start Menu
- On specific content using context-sensitive help via **F1**

dSPACE Help (Web) You can access the Web version of dSPACE Help at www.dspace.com/go/help.

To access the Web version, you must have a *mydSPACE* account.

PDF files You can access PDF files via the 📄 icon in dSPACE Help. The PDF opens on the first page.

Implementing Custom Function Blocks

Objective

To implement custom function blocks, you must add a custom function block type to the function library. There are some basic rules and steps you must apply for each custom function block type.

Where to go from here

Information in this section

Basics on Implementing Custom Function Blocks

Objective

Custom function blocks are a way of including user-specific code for ConfigurationDesk in a real-time application. You can add custom function block types to the function library via custom function files.

XML schema

An XML schema describes the structure of an XML document. It defines the elements and attributes to be used in the XML file. It also defines the hierarchical structure and the multiplicity of the elements. You can find the current and previous schema versions for ConfigurationDesk custom function XML files here:

```
<InstallationFolder>\ConfigurationDesk\Implementation\UserFiles\
```

With ConfigurationDesk Version 5.3, a new XML schema was defined for custom function blocks (`CustomFunctionTypeSchema_V3_1.xsd`). ConfigurationDesk lets you update custom function files from the last schema version (`CustomFunctionTypeSchema_V3`, introduced with ConfigurationDesk Version 4.3) to the new schema via the **Create Updated Custom Function Type XML** command.

Note

- Custom function blocks created according to previous schema versions are still available in your ConfigurationDesk application.
- You cannot downgrade an updated custom function file to the previous schema version. You are recommended to back up the existing files before the update.
- To have access to the newest features, you are recommended to create new custom function XML files according to the current XML schema version. To accesss these features in an existing custom function block, you have to update it.

For reference information on the XML elements that you can use in the custom function XML file according to the current XML schema, refer to Custom Function Block XML Element Reference on page 65.

Workflow to implement a custom function block

There are a number of steps to apply if you want to create a new custom function block. If you start by modifying existing custom function files, or if you want to change your custom function block later, you might only have to perform some of the following steps.

1. Create the custom function XML file according to the current XML schema:
   ```
   <InstallationFolder>\ConfigurationDesk\Implementation\
   UserFiles\CustomFunctionTypeSchema_V3_1.xsd
   ```
   For more information on defining the structure and elements of the XML file, refer to Defining Custom Function Block Elements on page 17. For rules and attributes for the elements of the XML file, refer to Custom Function Block XML Element Reference on page 65.

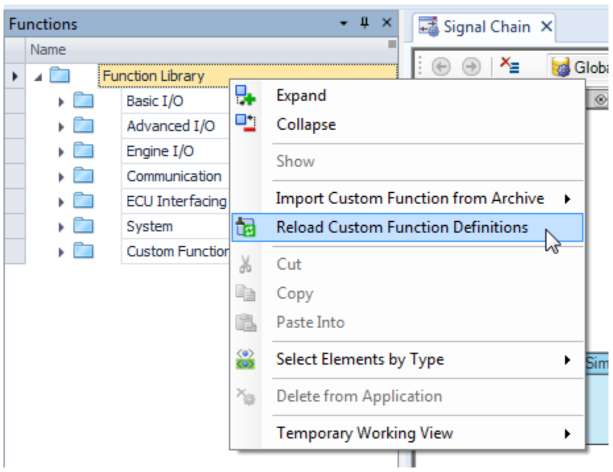
   Tip

   Use demo projects as a starting point or reference for your custom function block. Refer to Examples of custom function blocks on page 12.

2. Copy the XML file to the project-specific or global custom functions directory (refer to File Types and Directories for Custom Function Blocks on page 13).

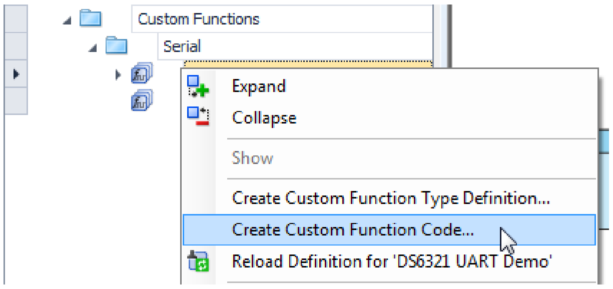
3. Integrate the custom function type in ConfigurationDesk. Choose one of the following alternatives:
   - If a ConfigurationDesk application is currently open:

     Reload the custom function block definitions via the **Reload Custom Function Definitions** command in the **Function Browser** context menu.

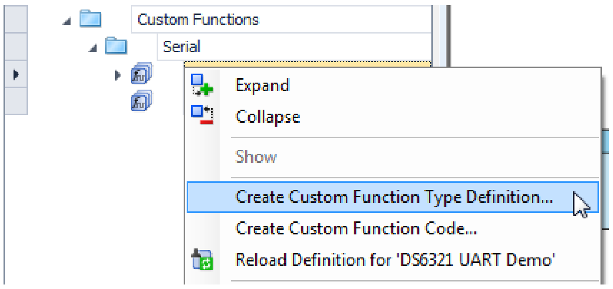


   - If no ConfigurationDesk application is currently open:

     Start ConfigurationDesk (if necessary) and create or open a project and an application.
4. Add the custom function block to the signal chain.
5. Generate the custom function code file templates via the **Create Custom Function Code** command in the **Function Browser** context menu.


Custom Functions
Serial
Expand
Collapse
Show
Create Custom Function Type Definition...
Create Custom Function Code...
Reload Definition for 'DS6321 UART Demo'

   ConfigurationDesk generates templates for the C++ source code file and the header file with predefined macros to access the elements defined in the XML file.
6. Generate the type definition file via the **Create Custom Function Type Definition** command in the **Function Browser** context menu.


Custom Functions
Serial
Expand
Collapse
Show
Create Custom Function Type Definition...
Create Custom Function Code...
Reload Definition for 'DS6321 UART Demo'

Note

The type definition file serves as a code reference for adjusting the source code files. Do not adjust it manually. Each build process causes the type definition file to be regenerated and overwritten.

7. Adjust the custom function source code files you generated in step 5 according to your needs. Use the type definition file you generated in step 6 as a reference when writing your own code.
8. Optional: Include user data (refer to Including User Data on page 35).
9. Reload the custom function block definitions.
10. Finish the application that contains the custom function block(s) and build the real-time application.

Note

- A version number (integer) has to be provided in the custom function XML file.
  See the following example from `UART.xml`:
  ```
  <CustomFunctionBlock Version="1" Name="UART">
    ...
  </CustomFunctionBlock>
  ```
  Each time the XML file is modified, you must increase its version number to apply the changes to existing custom function blocks in a ConfigurationDesk application. If you receive a new version of the XML file from an external source, make sure that its version number is higher than that of the currently used XML file. Do not store XML files of the same custom function type with different version numbers in custom function directories.
- If you create the source code file templates via **Create Custom Function Code** for an existing custom function, the code files are overwritten. Make sure to back up your files to avoid losing user-specific code.

Examples of custom function blocks

ConfigurationDesk includes some demo projects and applications that include examples of custom function blocks.

You can find the custom function files from the demo projects in the according project folders.

The following demo projects and applications are available:

- *Ethernet*: The **CfgEthernetDemo** project contains **Ethernet Send** and **Ethernet Receive** custom function blocks. For more information, refer to Example: Implementing a Custom Function Block for Ethernet Communication on page 51.
- *UART*: There are several demo projects and applications containing custom function blocks for implementing different types of UART serial communication. For more information, refer to UART Demo Projects and Applications (ConfigurationDesk UART Implementation 🕮).

Tip

Many of the examples in this document are taken from one of the demo projects.

File Types and Directories for Custom Function Blocks

Custom function files

A custom function block type consists of an XML file, two header files (.h files), a C++ source code file (CPP file), and optionally additional user-defined make artifacts.

XML file The XML file defines the fundamental properties of the custom function block type and its appearance in ConfigurationDesk.

The XML file is verified according to an XML schema, see:

```
<InstallationFolder>\ConfigurationDesk\Implementation\UserFiles\CustomFunctionTypeS
chema_<SchemaVersion>.xsd
```

For more information on defining the structure and elements of the XML file, refer to Defining Custom Function Block Elements on page 17. For structure information and attributes of the elements of the XML file, refer to Custom Function Block XML Element Reference on page 65.

Source code file and header files The functions of the C++ source code are published in the header files and implemented in the CPP file.

The names of the C++ source code files (without the extensions `.cpp`, `.h`, and `_TypeDef.h`) must be specified in the `CModule` element of the XML file. They must not contain whitespaces and other special characters.

Example from `UART_RS232_FlowControl.xml`:

```
<CModule Name="UART_RS232_FlowControl">
    ...
</CModule>
```

This specifies to implement the custom function block in the following files:

- `UART_RS232_FlowControl.h`
- `UART_RS232_FlowControl_TypeDef.h`
- `UART_RS232_FlowControl.cpp`

Once you created the XML file for your custom function block, you can automatically generate suitable source code and header files. When the files are generated, ConfigurationDesk uses the `CModule` name and adds an appropriate file extension.

Note

Simulink models used in the ConfigurationDesk application must not have the same name as any of the custom function files in the custom function directories. For example, you must not use `UART_RS232_FlowControl.mdl` as a model name while using the `UART RS232 FlowControl` custom function of the UARTRS32FlowControDemo project.

Additional make artifacts Apart from the basic custom function files, you can include additional make artifacts via the `MakeConfiguration` element in the XML file.

Example from `UART.xml`:

```
<MakeConfiguration>
  <Artefact Name="DsIoFuncUart.h" Type="Headerfile" />
  <Artefact Name="RES_GENERATED_DsIoFuncUart.h" Type="Headerfile" />
  <Artefact Name="dsuartdrv" Type="Library" />
</MakeConfiguration>
```

The make artifacts do not have to be located in a custom function directory. You can specify different directories via the `Directory` attribute.

Artifacts of types that are included in the real-time application file can be accessed via the `getArtifactsDirectory()` function in the type definition file (`<custom function nam>_TypeDef.h`).

For the XML reference information on including additional make artifacts, refer to Make Artifacts on page 120.

Custom function directories

To use a custom function block type in ConfigurationDesk, you must copy custom function files either to a project-specific custom functions directory or to the global custom functions directory. ConfigurationDesk browses both directories when a project is being loaded. All custom function block types are displayed in the Function Browser under the Custom Functions folder. You can specify an additional subfolder to categorize the custom function block types (refer to Defining a Custom Function Block Type on page 17).

The project-specific custom function directory is:

```
<ProjectRootDirectory>\<Project>\CustomFunctions
```

The default global search path for custom function files is:

```
<DocumentsFolder>\UserFiles
```

You can change the global search path on the **Configuration Page** page of the **ConfigurationDesk Options** dialog.

Note

- You must copy custom function files directly to the specified directory. Files from subdirectories are ignored.
- ConfigurationDesk lets you import custom function blocks from ZIP files via the **Import Custom Function from Archive** command.
- If ConfigurationDesk finds two custom function XML files in which the same custom function block type name is specified in the project-specific directory and the global search path, only the custom function from the project-specific directory is available in the project.
- Unlike custom function files from project-specific custom functions directories, custom function files from the global search path are not included in a project backup. You must provide them separately, for example to use them on a different system.

Defining Custom Function Block Elements

Objective

You can define all standard elements of function blocks, such as electrical interface units, signal ports, and properties, for your custom function block in the custom function XML file.

Where to go from here

Information in this section

Defining a Custom Function Block Type

Objective

To implement a custom function block, you must define a custom function block type.

Example for defining a custom function block type

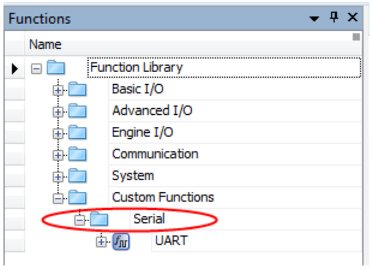
A custom function block type is defined via the `CustomFunctionType` element in the XML file, see the following example from `UART.xml`:

```
<?xml version="1.0"?>
<CustomFunctionType xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema" Version="3.1"
                    xmlns="urn:dSPACE:CustomFunctionTypeSchema">
  <Systemidentification>
    <SystemType Name="SCALEXIO" />
  </Systemidentification>
  <Classifications>
    <Classification Name="Serial" />
  </Classifications>
  <FunctionTypeFlags>
    <ExtendSignalChainOptions PortDataTypes="Inherited"/>
  </FunctionTypeFlags>
  <CustomFunctionBlock Version="1" Name="UART" DistributorInformation="dSPACE GmbH">

[...]

  </CustomFunctionBlock>
</CustomFunctionType>
```

Basic elements and attributes of a custom function block type

The following table describes the basic elements and attributes of a custom function block type. For detailed information on every possible element, refer to Custom Function Type on page 69.

XML Element	Description
`CustomFunctionType`	The root element for a custom function block type definition. The XML schema version must be provided via the **`Version`** attribute.
`Systemidentification`	Lets you specify which system the function block type is defined for in the `SystemType` element. Currently only the SCALEXIO system type is available.
`Classifications`	Lets you specify subfolders in the function library where the custom function block type is displayed. All custom function block types are displayed below the Custom Function folder in the Function Browser of ConfigurationDesk. If subfolders are specified, the custom

XML Element	Description
	function block type is displayed in them. The subfolders are specified via `Classification` elements. The illustration below shows a UART custom function block type for which one `<Classification Name="Serial" />` element has been defined:  Functions Name Function Library Basic I/O Advanced I/O Engine I/O Communication System Custom Functions Serial UART You can add more `Classification` elements to create a deeper hierarchy of subfolders. If you leave the `Name` string of the only `Classification` element empty, the custom function type will appear directly in the Custom Function folder.
`FunctionTypeFlags`	Lets you define the type-specific behavior for various aspects. ▪ The `ExtendSignalChainOptions` element lets you define how model port blocks will be generated via the 'Extend signal chain' operation. ▪ The `Scripting\Script` elements lets you define Python scripts that will be triggerable on each block instance via its context menu (in a working view).
`CustomFunctionBlock`	The element defining the actual parts of each function block instance: ▪ Electrical Interface on page 78 ▪ Model Interface on page 99 ▪ Hook Function Definitions on page 114 ▪ Make Artifacts on page 120 The name of the custom function block type as it is displayed in ConfigurationDesk is defined via the `Name` attribute.

Related topics

References

Custom Function Type.. 69

Defining the Electrical Interface Unit

Objective

An electrical interface unit provides a function block's interface to external devices and to the real-time hardware (via hardware resource assignment). It contains logical signals with signal ports. Signal ports represent the electrical connection points of a custom function block. They can also be used to generate wiring information for an external cable harness.

Example for defining electrical interface units

The electrical interface units of a custom function block are defined via the `PhysicalLayerInterfaces` element in the XML file as part of the `CustomFunctionBlock` element, see the following example from `UART.xml`:

```
<PhysicalLayerInterfaces>
  <PhysicalLayerInterface Id="Layer1" Name="UART">
    <LogicalSignals>
      <LogicalSignal Id="LogSig1" Name="KLine">
        <HwRequirements>
          <Resource Name="Data_Res" Id="Resource1" ResourceType ="LIN_1"/>
        </HwRequirements>
        <SignalPorts>
          <SignalPort Name="KLine_VBat" Direction="Reference" IsFiuEnabled="false" Id="SP1">
            <ResourceSignal Name="VBat" ResourceId="Resource1"/>
          </SignalPort>
          <SignalPort Name="KLine" Direction="Bidirectional" IsFiuEnabled="true" Id="SP2">
            <ResourceSignal Name="PinA" ResourceId="Resource1"/>
          </SignalPort>
          <SignalPort Name="KLine_Gnd" Direction="Reference" IsFiuEnabled="false" Id="SP3">
            <ResourceSignal Name="Reference" ResourceId="Resource1"/>
          </SignalPort>
        </SignalPorts>
      </LogicalSignal>
    </LogicalSignals>
  </PhysicalLayerInterface>
</PhysicalLayerInterfaces>
```

Basic elements and attributes of an electrical interface unit

The following table describes the basic elements and attributes of an electrical interface unit. For detailed information on every possible element of electrical interfaces, refer to Electrical Interface on page 78.

XML Element	Description
`PhysicalLayerInterface`	Lets you define an electrical interface unit for the function block type. An electrical interface unit provides an interface of the function block to the external devices and to the real-time hardware (via hardware resource assignment). It also provides properties to configure the characteristics of the hardware.
`LogicalSignal`	Lets you define a logical signal. A logical signal is used to group signal ports and relates to HW resources defined via `HwRequirements`.
`Resource`	Lets you define a channel request. A concrete channel request indicates which hardware properties are required and what pins are expected at the assigned channel. The name of the channel request will be displayed in configuration dialogs, e.g., the Properties Browser. The `ResourceType` attribute lets you define the channel type to be used for the channel request.
`SignalPort`	Lets you define a signal port. A signal port represents an electrical connection point of the function block. A signal port must only use `ResourceSignals` of hardware resources that are requested by its `LogicalSignal`. The `Direction` attribute defines the port direction. Signal ports with role "Reference" must also specify the direction "Reference". The `IsFiuEnabled` attribute defines if failure simulation functionality can be defined for the signal port.

XML Element	Description
`ResourceSignal`	Lets you define the requested signal type of the hardware system. Represents a reference to a pin of a defined hardware resource. You must only use `ResourceSignal` elements of hardware resources that are requested by the parent `LogicalSignal`.

Related topics

References

Defining the Model Interface

Objective

To connect the I/O functionality in ConfigurationDesk with the behavior model (e.g., a Simulink model), you need a model interface. This interface is implemented via model ports that are grouped in model port blocks. You can connect a signal to a model port in ConfigurationDesk by mapping a function port to it. The signal is then available at the corresponding model port in the behavior model and vice versa.

Functions and function ports

Within the model interface, function ports are grouped in functions. This allows for a consistent data exchange between all the function ports within one function. In the executable application, all the function ports of a function are triggered at the same time so that all the I/O data of the function outports in a function is consistent. All the function inports in a function must get consistent I/O data from the behavior model. I/O data must therefore be calculated in the same function module. For details on function modules and the timing of the I/O access, refer to Basics on Function Triggers (ConfigurationDesk Real-Time Implementation Guide 🕮).

Access points to function ports

Each function port has an access point that you can use to execute a user-defined C++ function of a custom function block. You must declare a hook function for each function port. You can also declare hook functions for whole functions or function blocks. For details, refer to Mandatory Hook Functions for Function Ports on page 33.

Example for defining a model interface

The model interface of a custom function block is defined via the `ModelInterface` element in the XML file as part of the `CustomFunctionBlock` element, see the following example from `UART.xml`:

```
<ModelInterface>
  <Functions>
    <Function Name="Transmit" Id="Fcn1" IsMultipleTriggerable="true">
      <DataPort Name="TxValue" Id="Fcn1_TxValue" Direction="In" DataType="UInt32" DefaultValue="0" Unit=""
      Width="1"/>
    </Function>
    <Function Name="Receive" Id="Fcn2" IsMultipleTriggerable="true">
      <DataPort Name="RxValue" Id="Fcn2_RxValue" Direction="Out" DataType="UInt32" DefaultValue="0" Unit=""
      Width="1"/>
    </Function>
  </Functions>
  <Parameters>
    <EnumerationParameter Name="Baudrate" Id="EnumParam1" AccessFlags="!Tunable" DataType="UInt32" DefaultIndex="5">
      <EnumerationValues>
        <EnumerationValue InternalValue="1200" DisplayName="1200" TraceValue="1200" />
        <EnumerationValue InternalValue="2400" DisplayName="2400" TraceValue="2400" />
        <EnumerationValue InternalValue="4800" DisplayName="4800" TraceValue="4800" />
        <EnumerationValue InternalValue="9600" DisplayName="9600" TraceValue="9600" />
        <EnumerationValue InternalValue="19200" DisplayName="19200" TraceValue="19200" />
        <EnumerationValue InternalValue="20000" DisplayName="20000" TraceValue="20000" />
      </EnumerationValues>
    </EnumerationParameter>
  </Parameters>
</ModelInterface>
```

Basic elements and attributes of a model interface

The following table describes the basic elements and attributes of the example. For detailed information on every possible element of the model interface, refer to Model Interface on page 99.

XML Element	Description
`ModelInterface`	Lets you define the model interface for the function block type.
`Function`	Lets you define a function providing an interface to the behavior model via model port blocks. A function can contain one or more data function ports (function signals) that are triggered together. You can specify whether multiple triggers are allowed or not.
`DataPort`	Lets you define a function port. You can map the function port to a model port. Several attributes such as the signal direction and the data type can be defined.
`Parameters`	You can define configuration properties for several elements of the custom function block type. Here, an enumeration property is defined for the `ModelInterface` element. For details on defining properties, refer to Defining the Configuration Properties on page 24.

Related topics

References

Model Interface.. 99

Defining I/O Events

Objective

I/O events are events that can be used to trigger tasks asynchronously by means of hardware events (e.g., hardware events of the UART driver).

Defining I/O events in the XML file

To use the I/O event mechanism to trigger tasks of executable applications, you must define each I/O event in the XML file as a part of the custom function block by means of an `EventPort` element. Example:

```
<ModelInterface>
    <EventPorts>
      <EventPort Name="TxFifoEmpty" Id="ep1"
      EventIdentifier="DsNIoFuncUart::Event::TxFifoEmpty" EventType="SW">
      </EventPort>

      ...

    </EventPorts>

    ...

</ModelInterface>
```

The example defines an I/O event named `TxFifoEmpty` with ID `ep1` and links it to the `TxFifoEmpty` event (transmit FIFO is empty) of the UART driver. See the `Event` namespace shown above. For details on the example, refer to Defining I/O Events (Serial Communication) on page 43.

This entry generates an I/O event in ConfigurationDesk which is provided by the custom function block. You can use the event to trigger tasks.

Generating events in the C++ source code file

The creation function in the C++ source code must declare a driver object for generating the events. Example:

```
void ucf_<CreateHookFunctionName>(DsTErrorList ErrorList,
                ucf_<CModuleName>Struct_T* pBlockInstance)
{
    [...]
    // Declare driver object as trigger for event (See xml-file for event code)
    pBlockInstance-><EventName>_driver = <DrvObject>
}
```

- `<CreateHookFunctionName>`: The name of the hook function for the creation access point as defined in the XML file.
- `<EventName>`: The name of the event as defined in the XML file.
- `<DrvObject>`: A driver object that can be created, for example, via the `<CModule>_DRIVER_CREATE` macro.

Related topics

Basics

Modeling Executable Applications and Tasks (ConfigurationDesk Real-Time Implementation Guide 📖)

References

Defining the Configuration Properties

Objective

Configuration properties can be changed via the Properties Browser when the custom function block is selected in ConfigurationDesk, without modifying the behavior model or the C++ source code.

Defining properties in the XML file

You can define configuration properties for several elements of the custom function block XML file (e.g., `CustomFunctionBlock`, `SignalPort`, `Function`).

See the following example for the `ModelInterface` element from `Ethernet_Send.xml`:

```
<ModelInterface>
  [...]
  <Parameters>
    <Parameter Name="Remote Address"
               Id="param1"
               AccessFlags="!Tunable"
               DataType="Char[]"
               DefaultValue="192.168.1.22"
               RangeMin=""
               RangeMax=""
               Description="The IP of the receiving system."/>
    <Parameter Name="Remote Port"
               Id="param2"
               AccessFlags="!Tunable"
               DataType="UInt16"
               DefaultValue="44000"
               RangeMin="1"
               RangeMax="65535"
               Description="The port of the receiving system."/>
    <EnumerationParameter Name="Protocol"
                          Id="param3"
                          AccessFlags="!Tunable"
                          DataType="UInt32"
                          DefaultIndex="1"
                          Description="Protocol to be used. (UDP (default) or TCP)">
      <EnumerationValues>
        <EnumerationValue InternalValue="0"
                          DisplayName="UDP"
                          TraceValue="0" />
        <EnumerationValue InternalValue="1"
                          DisplayName="TCP"
                          TraceValue="1" />
      </EnumerationValues>
    </EnumerationParameter>
  </Parameters>
</ModelInterface>
```

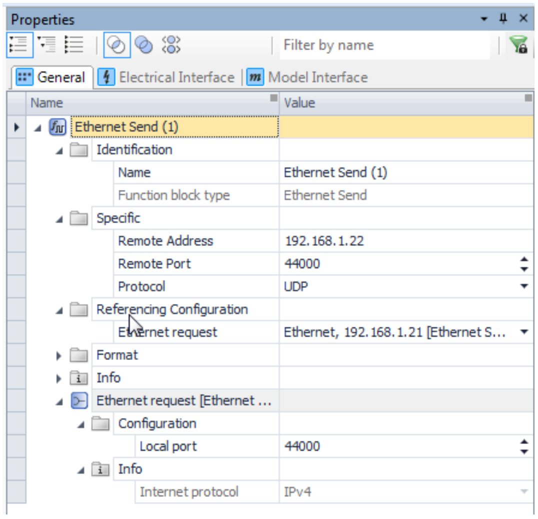
In your ConfigurationDesk application, properties defined for the `ModelInterface` element are available in the Properties Browser after selecting the custom function block.

The following table describes the basic elements and attributes of the example. For detailed information, refer to Properties on page 108.

XML Element	Description
`Parameter`	Lets you define a configuration property. A configuration property is defined by its name, data type and initial value. The configured value can be accessed in the user code via the instance-specific runtime struct that you can find in the type definition file. The `AccessFlags` attribute defines if the property should be accessible via experiment software. You can also use it to define if the property is editable and visible in ConfigurationDesk.
`EnumerationParameter`	Lets you define a property list. The `EnumerationValue` entries each define an item on the list. These values are interpreted as strings and used as inline parameters in the generated C++ source code. An enumeration parameter is defined by its name, data type and the list of all possible values. The configured value can be accessed in the C++ code via the instance-specific runtime struct that you can find in the type definition file.

XML Element	Description
	Example: `days = {Monday,1},{Tuesday,2},{Wednesday,3},{Thursday,4},…` In the Properties Browser, you select a value from the list (e.g., Monday). In the C++ code, you use a method such as `'setDay(1)'`.

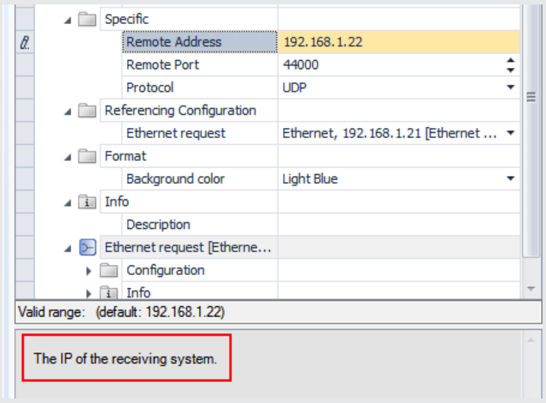
The following illustration shows the Properties Browser with specified configuration properties.



Name	Value
Ethernet Send (1)	
Identification	
Name	Ethernet Send (1)
Function block type	Ethernet Send
Specific	
Remote Address	192.168.1.22
Remote Port	44000
Protocol	UDP
Referencing Configuration	
Ethernet request	Ethernet, 192.168.1.21 [Ethernet S…
Format	
Info	
Ethernet request [Ethernet …	
Configuration	
Local port	44000
Info	
Internet protocol	IPv4

Tip

- The Specific property category is the default category for all custom function block properties you define. You can create other categories and assign properties to them by using the `Category` attribute of the `Parameter` or `EnumerationParameter` elements.
- The string provided in the `Description` property attribute is displayed in the help text area of the Properties Browser when the property is selected.



Specific	
Remote Address	192.168.1.22
Remote Port	44000
Protocol	UDP
Referencing Configuration	
Ethernet request	Ethernet, 192.168.1.21 [Ethernet …
Format	
Background color	Light Blue
Info	
Description	
Ethernet request [Etherne…	
Configuration	
Info	

Valid range: (default: 192.168.1.22)

The IP of the receiving system.

Using configuration properties in the C++ source code

A special macro is defined to read the values of configuration properties in the C++ source code:

```
<CustomFunctionBlockTypeName>_PARAMETER(<PropertyName>)
```

For example, to set the `Baudrate` property of the UART custom function block using the value specified in the Properties Browser of ConfigurationDesk, use the following code line:

```
pUartDrv->setBaudrate(ErrorList,UART_PARAMETER(Baudrate));
```

Related topics

References

Creating and Adjusting the Source Code Files

Objective

After defining the custom function block elements in the XML file, you must create the C++ source code files and adjust them according to your needs.

You can provide hook functions in your C++ source code files or include other code files, such as driver objects or library files, that are required for the build process.

Where to go from here

Information in this section

Information in other sections

Creating Source Code Templates

Objective

When you are defining a new custom function block type, you are recommended to use the source code template files created via **Create Custom Function Code** and **Create Custom Function Type Definition** commands in the **Function Browser**.

- Create Custom Function Code: ConfigurationDesk generates templates for the C++ source code file and the header file with predefined macros to access the elements defined in the XML file.
- Create Custom Function Type Definition: The type definition file serves as a code reference for adjusting the source code files. Each build process causes the type definition file to be regenerated and overwritten. Do not adjust it manually.

Related topics

Basics

Using Access Points to Trigger Hook Functions

Objective

An executable application has several access points that you can use to trigger C++ hook functions at specific times during the execution. For details on the timing of the I/O access, refer to Basics on Function Triggers (ConfigurationDesk Real-Time Implementation Guide 🕮).

Defining hook functions for access points

Hook functions for access points are defined in the `CModule` element of the custom function XML file. Access points are available at different levels:

- In the `CFunction-Global` element, you can define hook functions for access points concerning the whole custom function block.
- In the `CFunction-Trigger` element, you can define hook functions for access points concerning functions (`Function` elements) in the custom function block.
- In the `CFunction-DataPort` element, you can define hook functions for the access points for function ports in a function.

Note

A `CFunction-DataPort` element must be defined for each function port.

To specify when to execute the associated hook function, one of the elements above is used together with the embedded `Caller` element, which can have different attributes.

CFunction-Global The following table shows the attributes and values of a `Caller` element in a `CFunction-Global` element.

Attribute of Caller Element	Attribute Value	Description
`AccessPoint`	`Creation`	The associated hook function is used to create and initialize driver objects during the initialization of the executable application. Use this access point to allocate memory, for example.
	`OneTimeInit`	The associated hook function is called once after the executable application is started for the first time. Use this access point to set initial property values, for example.
	`Init`	The associated hook function is called each time the executable application is started. Use this access point to reinitialize objects or handle values from previous runs, for example.
	`Stop`	The associated hook function is called when the executable application is stopped. Use this access point to output termination values, for example.
	`Terminate`	The associated hook function is called when the executable application crashes. Use this access point to set termination values, for example.
	`Unload`	The associated hook function is called when the executable application is unloaded. Use this access point to free the allocated memory, for example.

CFunction-Trigger The following table shows the attributes and values of a `Caller` element in a `CFunction-Trigger` element.

Attribute of Caller Element	Attribute Value	Description
`AccessPoint`	`Entry`	The associated hook function is called at the beginning of the function module to which the function is assigned. It is called before the access points of the function ports. Use this hook function to make data available that must be consistent for the function ports, for example.
	`Exit`	The associated hook function is called at the end of a function module within the task to which the function is assigned. At this time, new data is available from the behavior model for the function inports and data can be sent to the I/O. If a function module is not executed, the access point is not reached and the associated hook function is not called.

Attribute of Caller Element	Attribute Value	Description
`FunctionId`	`<xyz>`	<xyz> is the ID of the hook function associated with the access point.

CFunction-DataPort The following table shows the attributes and values of a `Caller` element in a `CFunction-DataPort` element.

Attribute of Caller Element	Attribute Value	Description
`DataPortId`	`<xyz>`	<xyz> is the ID of the hook function associated with the access point.

Example of defining hook functions in the XML file

From `UART_RS232_FlowControl.xml`:

```
<CModule Name="UART_RS232_FlowControl">
    ...
    <CFunction-Global Name="Uart_Transmit_Init">
        <Caller AccessPoint="Init"/>
    </CFunction-Global>
    ...
    <CFunction-Trigger Name="Uart_Transmit_Exit">
        <Caller AccessPoint="Exit" FunctionId="Fcn1"/>
    </CFunction-Trigger>
</CModule>
```

Two hook functions are defined:

- A `CFunction-Global` hook function named `Uart_Transmit_Init` that is called at the start of the executable application.
- A `CFunction-Trigger` hook function named `Uart_Transmit_Exit` that is called when all the access points for the function ports of the function were executed. At this time, new data has been received for all the signals that get data from the connected behavior model.

Declaring hook functions in the header file

The hook functions for each access point must be declared in the header file. The following naming convention is mandatory:

```
void ucf_<AP-Name>([...]
             ucf_<CF-Name>Struct_T* pBlockInstance)
```

- `<AP-Name>` must be the name of the `CFunction` element for the access point,
- `<CF-Name>` must be the name of the `CModule`.

Example from`UART_RS232_FlowControl`:

```
void ucf_Uart_Transmit_Init(DsTErrorList ErrorList,
             ucf_UART_RS232_FlowControlStruct_T* pBlockInstance);
```

Mandatory Hook Functions for Function Ports

Objective

To exchange data with the behavior model, every function port has an access point (DataPort access point) and requires a hook function (DataPort function) that writes or reads data to or from the behavior model.

Read access points receive data from a behavior model (In direction). The read hook function is called when data from the behavior model is ready for the function port.

Write access points transfer data to the behavior model (Out direction). The write hook function writes the value to the data inport of the behavior model.

The following listing shows an example definition for a DataPort hook function:

```
<CModule Name="UART_RS232_FlowControl">
    ...
    <CFunction-DataPort Name="Uart_Transmit_TxBytes">
        <Caller DataPortId="Fcn1_TxBytes"/>
    </CFunction-DataPort>
    ...
</CModule>
```

The example code defines a hook function named `Uart_Transmit_TxBytes`. This DataPort hook function is called when the function port with the `Fcn1_TxBytes` ID (`TxBytes` function port) gets data from the behavior model.

Declaring a DataPort hook function in the header file

A DataPort hook function for each DataPort access point has to be declared in the header file. The following naming convention for the function name and the parameters is mandatory:

```
void ucf_<AP-Name> (ucf_<CF-Name>Struct_T* pBlockInstance,
           <DataType> * pValue);
```

- `<AP-Name>` must be the name of the `CFunction-DataPort` element.
- `<CF-Name>` must be the name of the `CModule`.
- `<DataType>` is the data type of the function port.

Implementing a DataPort hook function in the C++ source code

A DataPort hook function for each access point has to be implemented in the C++ source code file.

```
void ucf_<AP-Name> (ucf_<CF-Name>Struct_T* pBlockInstance,
<DataType> * pValue)
{
  // ...
}
```

- `<AP-Name>` must be the name of the `CFunction-DataPort` element.
- `<CF-Name>` must be the name of the `CModule`.
- `<DataType>` is the data type of the function port.

In a DataPort hook function of an In function port, data received from the behavior model is read from the `pValue` parameter and can be processed by the

DataPort function. In a DataPort function of an Out function port, data to be forwarded to the behavior model must be copied to the `pValue` parameter.

Example of a DataPort hook function from `UART_RS232_FlowControl`:

```
void ucf_Uart_Transmit_TxBytes(ucf_UART_RS232_FlowControlStruct_T*
                       pBlockInstance,  UInt8 * pValue)
{
  // ...
}
```

`Uart_Transmit_TxBytes` is the name of the hook function as defined in the XML file.

Defining and Implementing a Hook Function for a Global Access Point

Example of a hook function for a global access point

In the following example of a hook function for a global access point, driver objects are created and parameterized for the custom function block instance in the creation hook function.

Defining a hook function for the creation access point Example from `UART.xml`:

```
<CModule Name="UART">
    <CFunction-Global Name="Uart_Create">
        <Caller AccessPoint="Creation"/>
    </CFunction-Global>
...
</CModule>
```

Declaring the creation function In the header file, a function must be declared according to the following naming convention:

```
void ucf_<AP-Name>(DsTErrorList ErrorList,
       ucf_<CF-Name>Struct_T* pBlockInstance);
```

- `<AP-Name>` must be the name of the `CFunction-Global` element.
- `<CF-Name>` must be the `CModule` name from the XML file.
- The `ucf_<CF-Name>Struct_T*` structure is used to make all the information, such as trigger sources and configuration parameters, available, and to store information such as the pointer to the driver object.

Example from `UART.h`:

```
void ucf_Uart_Create(DsTErrorList ErrorList,
        ucf_UARTStruct_T* pBlockInstance);
```

Implementing the creation hook function The hook function has to be implemented in the C++ source code file. See the following example from `UART.cpp`:

```
/** <!------------------------------------------------------------------------->
*   Macro creates a UART driver object (DO NOT CHANGE!)
*
*   @parameters
*       @param P_DRIVER Pointer to the generated UART driver object (will be declared)
*       @param PHYS_INTERFACE Name of the PhysicalLayerInterface as declared in UART.xml
*
*   @note
*       To use in create function
*
*<!------------------------------------------------------------------------->*/
#define UART_DRIVER_CREATE(P_DRIVER,PHYS_INTERFACE)                                        \
            DsCIoFuncUart* P_DRIVER = DsCIoFuncUart::create(ErrorList, SimEngineApplGet());          \
            ((UART_INSTANCE_STRUCT*) (pBlockInstance->PHYS_INTERFACE.rtObjects))->pDrv = P_DRIVER ;        \


// ...


void ucf_Uart_Create(DsTErrorList ErrorList,
                  ucf_UARTStruct_T* pBlockInstance)
{

   // ...

   // Create driver for each instance
   UART_DRIVER_CREATE(pUartDrv,UART);

   // ...

}
```

In the example above, the driver object is created by using the pre-defined macro `UART_DRIVER_CREATE`. The parameters are the pointer to the driver object (P_DRIVER: `pUartDrv`) and the name of the electrical interface unit (PHYS_INTERFACE: `UART`).

Including User Data

Objective In the C++ source code of a custom function block, you can include user-specific data for each instance of the function block.

Declaring user data

You must declare user data at the beginning of the user code in the C++ source code. A user data structure named `<CModule name>_USER_DATA_STRUCT` is included in the generated source code file templates.

See the following example from `UART.cpp`:

```
/** <!------------------------------------------------------------------->
*   User data structure
*
*   @description
*     The following structure may be changed by the user. For every
*     instance
*     of the CustomIoFunction a buffer of this structure will be created by
*     the UART_INSTANCE_BUFFER_CREATE macro.
*
*<!------------------------------------------------------------------->*/
struct UART_USER_DATA_STRUCT  {
  UInt8 TxFrame[256];      // Buffer for transmit data frame
  UInt8 RxFrame[256];      // Buffer for receive data
  UInt32 NumTxBytes;       // Number of bytes to be sent
  UInt32 NumRxBytes;       // Number of received bytes
};
```

Creating user data

The macro normally creates the user data structure in the creation function of the custom function block. See the following example from `UART.cpp`:

```
UART_INSTANCE_BUFFER_CREATE(pUserData,<ELECT_INTERFACE>)
```

The macro creates a data structure of the UART_USER_DATA_STRUCT type. It returns a pointer to the created data structure via the `pUserData` variable. You can fill the user data structure with default data. See the following example from `UART.cpp`:

```
// Create user data buffer for each instance ...
UART_INSTANCE_BUFFER_CREATE(pUserData,UART);

// ... and set default values for some user variables
memset(pUserData->TxFrame,0,256);
memset(pUserData->RxFrame,0,256);
pUserData->NumTxBytes = 0;
pUserData->NumRxBytes = 0;
```

Using user data in other C++ functions

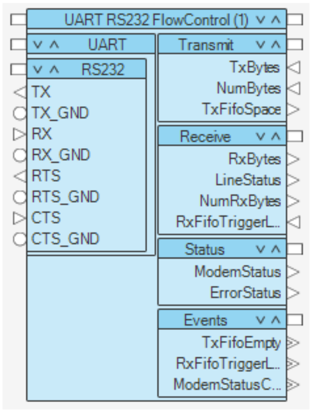
In the other C++ functions of the source code, you can get a pointer to the user data structure of the custom function block instance with the following macro:

```
// Get pointer to user data structure of instance
UART_INSTANCE_BUFFER_GET(pUserData,<ELECT_INTERFACE>);
```

Example: Implementing a Custom Function Block for UART Serial Communication

Objective

Serial communication must be implemented in the C++ programming language and a custom function block for ConfigurationDesk must be defined to integrate the serial communication.



Where to go from here

Information in this section

Basics on Implementing a Custom Function Block for UART Serial Communication

Modeling a UART serial communication

The protocol for UART serial communication is not available as a standard function block in ConfigurationDesk, because there is a wide range of different protocols and their variants. To implement the protocol, dSPACE provides a UART driver and the corresponding application programming interface (API). The programming language of the API is C++. You can use the API to create and configure UART driver objects within a custom function block and send or receive data and status information.

For more information, refer to ConfigurationDesk UART Implementation 🕮.

UART demos

Examples of UART custom function blocks are included in the UART demos of your dSPACE installation. They demonstrate how you can implement a custom function block for serial communication.

Tip

The custom function files from the UART demos can serve as a starting point for implementing your own custom function block.

The examples used in this document are based on the UARTAppl application in the CfgUARTDemo project and the UARTRS232FlowControlAppl application in the CfgUARTRS232FlowControlDemo project. For detailed descriptions of these demos and for additional UART demo descriptions, refer to UART Demo Projects and Applications (ConfigurationDesk UART Implementation 🕮).

UARTAppl

- Shows the basic functionality
- Uses a LIN/K-Line transceiver
- Required hardware resource: DS2680 I/O Unit with DS2672 Bus Module

The UART custom function block can be used with minimum external wiring. Only an external 12 V supply is required to simulate the battery voltage.

UARTRS232FlowControlAppl

- Uses flow control mechanisms (FC)
- Uses an RS232 transceiver
- Uses I/O events
- Consistent data transmission with multiple function signals per function block
- Required hardware resource: DS2671 Bus Board

To use the code example in the experiment software, the RS232 driver requires external wiring (RX/TX, RTS/CTS bridges and GND connections).

The demo examples define macros that hide complex internal details. The macros have certain limitations that might not be acceptable for some applications (for example, one driver object per custom function instance). That is why these macros are part of the examples and the generated templates `<CustomFunction>.cpp` and are not kept in a central location. If the macros must be adapted, copy the examples before changing them.

Generating a UART driver object

A custom function block for serial communication requires a UART driver object.

The UART driver object must be generated in the creation function of the C++ source code. Example from `UART.cpp`:

```
// Create driver for each instance
    UART_DRIVER_CREATE(pUartDrv, UART);
```

This macro generates a driver for the particular instance. `pUartDrv` is a pointer to the created driver object. `UART` is the electrical interface unit.

The macros defined in the example custom function block do not allow the generation of more than one driver object per instance of a custom function block.

Specifying basic settings Basic UART driver settings are specified in the creation function. Example from `UART.cpp`:

```
// Set up transceiver for sender and receiver
pUartDrv->setTransceiver(ErrorList,
          DsNIoFuncUart::Transceiver::LIN);
```

Note that the `DsNIoFuncUart::Transceiver::LIN` constant is used to select the LIN/K-Line transceiver for the custom function block.

Pointer to driver object In the other functions of the C++ source code, you can get a pointer to the UART driver object of the instance by means of the following macro:

```
// Get pointer to driver
UART_DRIVER_GET(pUartDrv,UART);
pUartDrv->setTransceiver(ErrorList, DsNIoFuncUart::Transceiver::LIN);
pUartDrv->applySettings(ErrorList);
```

You should also note that driver settings outside of the creation function can only take effect if you call the `applySettings` function after using the set methods of a driver.

Defining the Electrical Interface Unit (Serial Communication)

General information

- For basic information on the settings and structure of electrical interface units of custom function blocks, refer to Defining the Electrical Interface Unit on page 19.
- For a complete reference of the relevant XML elements for electrical interface units of custom function blocks, refer to Electrical Interface on page 78.

ResourceSignal settings

The connections to the internal bus channels are defined by the `ResourceSignal` elements of the XML file. This information is also important for calculating the wiring information for the external cable harness.

For a DS2671 Bus Board, the following pins (`Name` attribute of the `ResourceSignal` element) are available if the RS232 transceiver is selected (if another transceiver is selected, refer to Signal Mapping of the DS2671 Bus Board (SCALEXIO Hardware Installation and Configuration 🕮)):

- `PinA`: TX signal
- `PinB`: Ground for TX signal
- `PinC`: RX signal
- `PinD`: Ground for RX signal

For a DS2672 Bus Module, the following pins are available:

- `PinA`: LIN/K-Line signal
- `VBat`: Power supply
- `Reference`: Ground

HwRequirements settings

Example from `UART_RS232_FlowControl.xml`:

```
<PhysicalLayerInterfaces>
  <PhysicalLayerInterface Name="UART" Id="sc1">
    <LogicalSignals>
      <LogicalSignal Name="RS232" Id="LogSig1">
      <HwRequirements>
        <Constraint Name="ConsecutiveChannels">
          <ResourceReference ResourceId="Resource1"/>
          <ResourceReference ResourceId="Resource2"/>
        </Constraint>
        <Resource Name="Data_Res" Id="Resource1" ResourceType="Bus_1"/>
        <Resource Name="Handshake_Res" Id="Resource2" ResourceType="Bus_1"/>
      </HwRequirements>

        ...

    </LogicalSignal>
  </LogicalSignals>
  </PhysicalLayerInterface>
</PhysicalLayerInterfaces><
```

Using consecutive hardware channels The constraint `ConsecutiveChannels` specifies that the following two requested hardware resources must be covered by consecutive channels. In the `UART_RS232_FlowControl` example, this is the prerequisite for executing hardware flow control with RX/TX (first channel for data exchange named `Data_Res`) and RTS/CTS (second channel for flow control/handshake, named `Handshake_Res`). For details, refer to Creating and Adjusting the Source Code Files (Serial Communication) on page 46.

Addressing hardware resources In the XML file, `ResourceId` is used as a reference. In the example above, two resources are specified: `Resource1` (name: `Data_Res`) and `Resource2` (name: `Handshake_Res`).

In the C++ source code file, the name is used as a reference for binding the I/O resource to a UART driver object.

Specifying channel type The channel type is specified in the `ResourceType` attribute of the `Resource` element. The following channel types are possible for a UART driver.

- Bus_1: This channel type is on a DS2671 Bus Board. It can be used for UART and several bus types (CAN, LIN, FlexRay). It is used in the `UART_RS232_FlowControl` example.
- LIN_1: This channel type is on a DS2672 Bus Module (an optional module of the DS2680 I/O Unit). It can be used for UART as it has a UART controller and LIN/K-Line transceiver. It is used in the `UART` example.

Related topics

Basics

Defining the Electrical Interface Unit.. 19

References

Electrical Interface.. 78
Signal Mapping of the DS2671 Bus Board (SCALEXIO Hardware Installation and Configuration 📖)
Signal Mapping of the DS2672 Bus Module (SCALEXIO Hardware Installation and Configuration 📖)

Defining the Model Interface (Serial Communication)

General information

- For basic information on the model interface of custom function blocks, refer to Defining the Model Interface on page 21.
- For a complete reference of the relevant XML elements for the model interface of custom function blocks, refer to Model Interface on page 99.

Defining the model interface for UART_RS232_FlowControl

```
<ModelInterface>

...

  <Functions>
        <Function Name="Transmit" Id="Fcn1" IsMultipleTriggerable="true">
          <DataPort Name="TxBytes" Id="Fcn1_TxBytes" Unit="" Width="256" Direction="In" RangeMin="0" RangeMax="255"
          DefaultValue="0" DataType="UInt8" />
          <DataPort Name="NumBytes" Id="Fcn1_NumBytes" Unit="" Width="1" Direction="In" RangeMin="0" RangeMax="255"
          DefaultValue="0" DataType="UInt8" />
          <DataPort Name="TxFifoSpace" Id="Fcn1_TxFifoSpace" Unit="" Width="1" Direction="Out" RangeMin="0"
          RangeMax="4294967295" DefaultValue="0" DataType="UInt32" />
        </Function>
        <Function Name="Receive" Id="Fcn2" IsMultipleTriggerable="true">
          <DataPort Name="RxBytes" Id="Fcn2_RxBytes" Unit="" Width="256" Direction="Out" RangeMin="0" RangeMax="255"
          DefaultValue="0" DataType="UInt8" />
          <DataPort Name="LineStatus" Id="Fcn2_LineStatus" Unit="" Width="1" Direction="Out" RangeMin="0"
          RangeMax="255" DefaultValue="0" DataType="UInt8" />
          <DataPort Name="NumRxBytes" Id="Fcn2_NumRxBytes" Unit="" Width="1" Direction="Out" RangeMin="0"
          RangeMax="255" DefaultValue="0" DataType="UInt8" />
          <DataPort Name="RxFifoTriggerLevel" Id="Fcn2_RxFifoTriggerLevel" Unit="" Width="1" Direction="In"
          RangeMin="1" RangeMax="4294967295" DefaultValue="1" DataType="UInt32" />
        </Function>
        <Function Name="Status" Id="Fcn3" IsMultipleTriggerable="true">
          <DataPort Name="ModemStatus" Id="Fcn3_ModemStatus" Unit="" Width="1" Direction="Out" RangeMin="0"
          RangeMax="255" DefaultValue="0" DataType="UInt8" />
          <DataPort Name="ErrorStatus" Id="Fcn3_ErrorStatus" Unit="" Width="1" Direction="Out" RangeMin="0"
          RangeMax="255" DefaultValue="0" DataType="UInt8" />
    </Function>

    ...

  </Functions>

  ...

</ModelInterface>
```

In the example, the `Transmit` function is created with the `Fcn1` ID. The function is declared as multi-triggerable, which means you can use it in several function modules or tasks. However, you must ensure that the I/O data of the function is consistent. For example, the number of bytes to be sent (`TxBytes`) must be equal to the value which is given by `NumBytes`.

The `TxBytes` function port is created with the following properties:

- Id: 'Fcn1_TxBytes', internal ID in the XML file, which is used to reference this function port.
- Unit: No physical unit has been defined.
- Width: A vector or array with 256 elements of the specified data type.
- Direction: In (from the behavior model to the custom function), allowed values: `In` and `Out`
- RangeMin and RangeMax: The range for each element of the array is defined as 0 ... 255.

- DefaultValue: The default value of the function port. The value is used until the function port gets its value from the custom function code or behavior model.
- Data type: UInt8, allowed values: Int8, Int16, Int32, UInt8, UInt16, UInt32, Bool, Float32, Float64, dsfloat

Related topics

Basics

References

Defining I/O Events (Serial Communication)

General information

- For basic information on I/O events in custom function blocks, refer to Defining I/O Events on page 23.
- For a complete reference of the relevant XML elements for I/O events in custom function blocks, refer to Model Interface on page 99.

Available UART events

You can use different UART events to trigger tasks of an executable application. The event identifier can be found in the header file of the UART driver (`DsIoFuncUart.h`, see DsNIoFuncUart::Event (ConfigurationDesk UART Implementation 📖)):

```
/** <!------------------------------------------------------------------------->
*
*   @namespace DsNIoFuncUart::Event
*
*   @brief
*     Constants for UART events that may trigger application tasks.
*
*<!------------------------------------------------------------------------->*/
namespace Event
{
  const UInt32 LineStatusChanged      =   0x00000000U;
                /**< Event: Line status changed*/
  const UInt32 ModemStatusChanged     =   0x00000001U;
                /**< Event: Modem status changed*/
  const UInt32 RxFifoLevelReached     =   0x00000002U;
                /**< Event: Receive FIFO trigger level reached*/
  const UInt32 TxFifoEmpty            =   0x00000003U;
                /**< Event: Transmit FIFO empty*/
};
```

Defining I/O events in the XML file

To use the I/O event mechanism to trigger tasks of executable applications, you must define each I/O event in the XML file as a part of the custom function block by means of an `EventPort` element. Example:

```
    <ModelInterface>
      <EventPorts>
        <EventPort Name="TxFifoEmpty" Id="ep1"
         EventIdentifier="DsNIoFuncUart::Event::TxFifoEmpty" EventType="SW">

        </EventPort>

        ...

      </EventPorts>

      ...

</ModelInterface>
```

The example defines an I/O event named `TxFifoEmpty` with ID `ep1` and links it to the `TxFifoEmpty` event (transmit FIFO is empty) of the UART driver. See the `Event` namespace shown above.

This entry generates an I/O event in ConfigurationDesk which is provided by the custom function block. The event can be used to trigger tasks.

Generating events in the C++ source code file

The creation function of the C++ source code must define the UART driver object for generating the events. The following macro can be used:

```
// Declare driver object as trigger for transmit event (See xml-file for event
code)
UART_DRIVER_TRIGGER_EVENT(pUartDrv,TxFifoEmpty)
```

In this example, the `pUartDrv` driver object is defined as the source for the `TxFifoEmpty` I/O event, which is described in the `EventPorts` element of the XML file.

Related topics

Basics

Defining I/O Events...23
Modeling Executable Applications and Tasks (ConfigurationDesk Real-Time Implementation Guide 🕮)

References

Model Interface...99

Defining the Configuration Properties (Serial Communication)

General information

- For basic information on configuration properties in custom function blocks, refer to Defining the Configuration Properties on page 24.
- For a complete reference of the relevant XML elements for configuration properties in custom function blocks, refer to Properties on page 108.

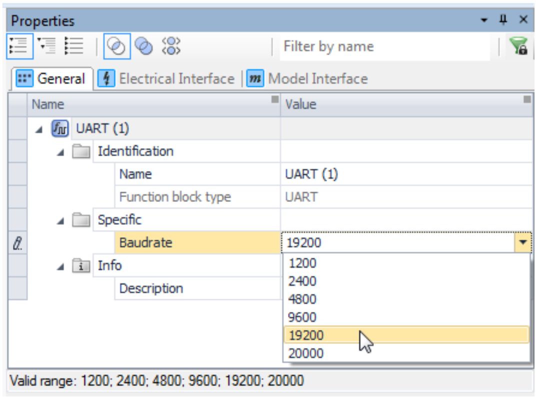
Defining properties in the XML file

Configuration properties can be specified for the `CustomFunctionBlock` element in the XML file, e.g., as part of the `ModelInterface` element. Example from `UART.xml`:

```
<CustomFunctionBlock>
    ...
    <ModelInterface>
    ...
        <Parameters>
            <EnumerationParameter Name="Baudrate" Id="EnumParam1" AccessFlags="!Tunable" DataType="UInt32"
            DefaultIndex="5"
             >
                <EnumerationValues>
                    <EnumerationValue InternalValue="1200" DisplayName="1200" TraceValue="1200" />
                    <EnumerationValue InternalValue="2400" DisplayName="2400" TraceValue="2400" />
                    <EnumerationValue InternalValue="4800" DisplayName="4800" TraceValue="4800" />
                    <EnumerationValue InternalValue="9600" DisplayName="9600" TraceValue="9600" />
                    <EnumerationValue InternalValue="19200" DisplayName="19200" TraceValue="19200" />
                    <EnumerationValue InternalValue="20000" DisplayName="20000" TraceValue="20000" />
                </EnumerationValues>
            </EnumerationParameter>
        </Parameters>
    </ModelInterface>
    ...
</CustomFunctionBlock>
```

In this example, a `Baudrate` property of the UInt32 type is defined with ID `EnumParam1`. Because the value of `AccessFlags` is `!Tunable`, the property is not included in the TRC file. It is therefore not possible to tune this property with experiment software. The property can be set via a list in ConfigurationDesk (`EnumerationParameter`). The `EnumerationValue` entries each define an item on the list. These values are interpreted as strings and are used as inline parameters in the generated code. It is therefore also possible to list namespace constants of header files. See the `UART_RS232_FlowControl.xml` as an example.

The following illustration shows the Properties Browser with the specified configuration properties.


Properties

Filter by name

General | Electrical Interface | Model Interface

Name	Value
UART (1)	
Identification	
Name	UART (1)
Function block type	UART
Specific	
Baudrate	19200
Info	
Description	

1200
2400
4800
9600
19200
20000

Valid range: 1200; 2400; 4800; 9600; 19200; 20000

Using configuration properties in the C++ source code

A special macro is defined to read the values of configuration properties in the C++ source code:

```
<CustomFunctionBlockTypeName>_PARAMETER(<PropertyName>)
```

For example, to set the `Baudrate` property of the UART custom function block using the value specified in the Properties Browser of ConfigurationDesk, use the following code line:

```
pUartDrv->setBaudrate(ErrorList,UART_PARAMETER(Baudrate));
```

Related topics

Basics

Defining the Configuration Properties... 24

References

DsCIoFuncUart::setBaudrate (ConfigurationDesk UART Implementation 📖)
Properties.. 108

Creating and Adjusting the Source Code Files (Serial Communication)

General information

- For basic information on the source code files of custom function blocks, refer to Creating and Adjusting the Source Code Files on page 29.
- For a complete reference of the relevant XML elements in custom function blocks, refer to Hook Function Definitions on page 114.

Binding the UART driver to several functions

To enable data exchange between driver objects and the function of the custom function block, including its function ports, you have to bind the driver objects to a function module. The driver and the behavior model exchange data only if a driver object is bound to the function module.

You can bind the driver object and the function by using a macro in the C++ source code:

```
// Bind driver to Transmit function
UART_DRIVER_BIND_FUNCTION(pUartDrv,Transmit);
```

In the example, the `pUartDrv` driver object is bound to the `Transmit` function of the custom function block. If the driver has to exchange data with several subsystems of the behavior model, you can bind it several times.

Note

If you bind the driver several times, make sure that

- This is supported by the driver.
- This does not lead to inconsistent data.

The function module receives data from the driver before being executed. The function module sends data to the driver after being processed.

Binding hardware resources to driver objects

In the creation function of the C++ source code, the requested hardware resources are linked to the generated driver objects. Example from `UART.cpp`:

```
// Bind hardware resource (channel) to driver
// 3rd Parameter: 0: data channel, 1: handshake channel
// 4th Parameter: 1: master channel (may trigger events),
//                0: slave channel (may not trigger events)
UART_DRIVER_BIND_CHANNEL(pUartDrv,Data_Res,0,0)
```

This example shows how the `Data_Res` hardware resource is bound to the `pUartDrv` driver object as a data channel. The resource acts as a slave channel and cannot trigger events.

Several channels of a hardware used by one driver object

To implement flow control for a UART, two channels must be used by one UART driver object because one channel of a DS2671 Bus Board does not feature enough (external) connector pins. Flow control is only possible on a DS2671 Bus Board, not on a DS2672 Bus Module of the DS2680 I/O Unit.

Using consecutive channels Each channel of the DS2671 Bus Board has four bus lines that have a different functionality depending on the protocol and transceiver. In some situations the four bus lines might not be enough to realize all necessary bus signals in a connection.

For example, with a hardware handshake with a RS232 transceiver, RTS/CTS handshake signals are required in addition to the ground and the RXD and TXD data lines, so that altogether at least five lines are required. To solve this problem, a bus channel can share the line of an adjacent channel. In order to improve immunity against signal disturbance, each signal line has an associated ground line. Thus a driver object uses two channels of the bus board. One channel is the data channel serving the RXD and TXD lines. The second channel is the handshake channel realizing the RTS/CTS lines.

Defining consecutive channels in the XML file To use consecutive channels, two adjacent bus channels have to be requested from the XML file as resources:

```
<PhysicalLayerInterfaces>
  <PhysicalLayerInterface Id="sc1" Name="UART">
    <LogicalSignals>
      <LogicalSignal Id="LogSig1" Name="RS232">
      <HwRequirements>
        <Constraint Name="ConsecutiveChannels">
          <ResourceReference ResourceId="Resource1"/>
          <ResourceReference ResourceId="Resource2"/>
        </Constraint>
        <Resource Name="Data_Res" Id="Resource1" ResourceType="Bus_1"/>
        <Resource Name="Handshake_Res" Id="Resource2" ResourceType="Bus_1"/>
      </HwRequirements>
    </LogicalSignal>
  </LogicalSignals>
  </PhysicalLayerInterface>
</PhysicalLayerInterfaces>
```

In this example from the `UART_RS232_FlowControl.xml` file, the `Data_Res` and `Handshake_Res` resources are requested as adjacent channels (`ConsecutiveChannels`).

Binding channels to a driver object In the creation function in the C++ source code, the two channels are bound to the `pUartDrv` driver object:

```
// Bind first hardware resource (channel) to driver: data channel
// Parameter: 0: data channel, 1: master channel (may trigger events)
UART_DRIVER_BIND_CHANNEL(pUartDrv,Data_Res,0,1)
// Bind second hardware resource (channel) to driver: handshake channel
// Parameter: 1: handshake channel, 0: slave channel (may not trigger
// events)
UART_DRIVER_BIND_CHANNEL(pUartDrv,Handshake_Res,1,0)
```

This driver object affects both channels in subsequent configuration settings:

```
// Set up transceiver
pUartDrv->setTransceiver(ErrorList,DsNIoFuncUart::Transceiver::RS232);
```

This function sets the RS232 transceiver for both channels after the `applySettings` method is called. Thus RXD and TXD, and RTS and CTS are operated with RS232 level.

Defining and implementing a hook function for a global access point

The following example shows the typical use of an access point of the init type:

```
void ucf_Uart_Transmit_Init(DsTErrorList ErrorList,
        ucf_UART_RS232_FlowControlStruct_T* pBlockInstance)
{

    // Get pointer to driver
    UART_DRIVER_GET(pUartDrv,UART);

    // Start send driver
    pUartDrv->start(ErrorList);
}
```

The UART driver is started when the executable application starts. After initialization, the UART driver starts sending and receiving data and creating events (if they are configured).

The following example shows the typical use of an access point of the function trigger exit type:

```
void ucf_Uart_Transmit_Exit(DsTErrorList ErrorList,
           ucf_UART_RS232_FlowControlStruct_T* pBlockInstance)
{
   // Local variable for sent bytes
   UInt32 NumTxBytes;

   // Get pointer to user data structure of instance
   UART_INSTANCE_BUFFER_GET(pUserData,UART);

   // Get pointer to driver
   UART_DRIVER_GET(pUartDrv,UART);

   // Send data frame
   pUartDrv->send(ErrorList, pUserData->NumBytes,
        (UInt8*) (pUserData->TxBytes), &NumTxBytes);
}
```

The hook function allows consistent processing of all data that was sent by the behavior model.

Related topics

Basics

Basics on Function Triggers (ConfigurationDesk Real-Time Implementation Guide 🕮)
Defining the Electrical Interface Unit (Serial Communication)...39

References

DsCIoFuncUart::applySettings (ConfigurationDesk UART Implementation 🕮)
DsCIoFuncUart::bindIoChannel (ConfigurationDesk UART Implementation 🕮)
DsCIoFuncUart::setTransceiver (ConfigurationDesk UART Implementation 🕮)
DsNIoFuncUart::ChannelType (ConfigurationDesk UART Implementation 🕮)
Hook Function Definitions..114

Example: Implementing a Custom Function Block for Ethernet Communication

Objective

Ethernet communication must be implemented in the C++ programming language and a custom function block for ConfigurationDesk must be defined to integrate the Ethernet communication.

Where to go from here

Information in this section

Basics on Implementing a Custom Function Block for Ethernet Communication

Basics on Implementing a Custom Function Block for Ethernet Communication

Required knowledge

To be able to implement an Ethernet communication, you have to be familiar with network communication and configuration.

Ethernet demo

Two examples of Ethernet custom function blocks (**Ethernet Send** and **Ethernet Receive**) are included in the Ethernet demo in your dSPACE installation. They give you basic information on exchanging data with a second Ethernet port in your SCALEXIO or MicroAutoBox III system.

Tip

- The custom function files from the Ethernet demo can serve as a starting point for implementing your own custom function block.
- ConfigurationDesk also offers Ethernet function blocks that are ready to use. Refer to Ethernet (ConfigurationDesk I/O Function Implementation Guide 📖).

Ethernet Send

- Shows the basic functionality
- Sends via a UDP or TCP connection
- Establishes a TCP connection to another system
- A second Ethernet port is required.

Ethernet Receive

- Receives via a UDP or TCP connection
- For TCP connections: waits for an incoming connection after the application starts in a separate thread
- Receives one UInt32 value and provides it to the model
- A second Ethernet port is required.

To use the examples, there must be a connection to the host setup on a separate network. The following default network parameters are set in the demo:

- Host IP 192.168.1.22
- SCALEXIO IP on second interface 192.168.1.21
- Subnet Mask 255.255.255.0

Note

The Ethernet demo custom function blocks support only IPv4. Configuring IPv6 at the **Ethernet Setup** function block causes an error message when executing the real-time application.

The demo examples define macros that hide complex internal details. The macros have certain limitations that might not be acceptable for some applications. That is why these macros are part of the examples and of the generated `<CustomFunction.CPP>` template, and are not kept in a central location. If the macros must be adapted, copy the examples before changing them.

Related topics

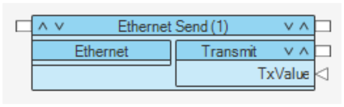
Basics

Ethernet (ConfigurationDesk I/O Function Implementation Guide 🕮)

Defining a Custom Function Block Type for Ethernet Communication

Objective

To use a custom function block with ConfigurationDesk, you must specify a custom function block type. You can configure the custom function blocks of this type and use them in the logical signal chain. The custom function block type is specified in the custom function XML file.



Where to go from here

Information in this section

Defining the Model Interface (Ethernet)

General information

- For basic information on the model interface of custom function blocks, refer to Defining the Model Interface on page 21.
- For a complete reference of the relevant XML elements for the model interface of custom function blocks, refer to Model Interface on page 99.

Defining the model interface for Ethernet_Send

```
<ModelInterface>
    <Functions>
        <Function Name="Transmit"
                  Id="Fcn1"
                  IsMultipleTriggerable="true">

            <DataPort Name="TxValue"
                      Id="Fcn1_TxValue"
                      Unit=""
                      Width="1"/>
                      Direction="In"
                      RangeMin="0"
                      RangeMax="4294967295"
                      DefaultValue="0"
                      DataType="UInt32"/>
        </Function>
    </Functions>
    [...]
</ModelInterface>
```

In the example, the `Transmit` function is created with the `Fcn1` ID. The function is declared as multi-triggerable, which means you can use it in several tasks. However, you must ensure that the I/O data of the function is consistent.

The `TxValue` function port is created with the following properties:

- Id: 'Fcn1_TxValue', internal ID in the XML file, which is used to reference this function port.
- Unit: A physical unit has not been defined.
- Width: A scalar value with 1 element of the specified data type is expected.
- Direction: In (from the behavior model to the custom function), allowed values: `In` and `Out`
- RangeMin and RangeMax: The value range is defined as 0 ... 4294967295.
- DefaultValue (=0): This property is used as the initial value, i.e., if the function port has not yet received any data from the custom function code (outports) or from the behavior model (model ports).
- Data type: UInt32, allowed values: Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32, UInt64, Bool, Float32, Float64, dsfloat

Related topics

Basics

Defining the Model Interface.. 21

References

Model Interface.. 99

Defining the Configuration Properties (Ethernet)

General information

- For basic information on configuration properties in custom function blocks, refer to Defining the Configuration Properties on page 24.
- For a complete reference of the relevant XML elements for configuration properties in custom function blocks, refer to Properties on page 108.

Defining properties in the XML file for Ethernet_Send

Configuration properties can be specified for the `CustomFunctionBlock` element in the XML file, for example, as part of the `ModelInterface` element. Example from `Ethernet_Send.xml`:

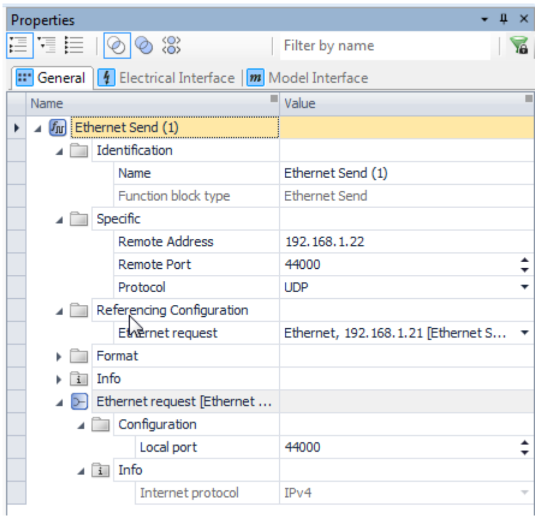
```
<ModelInterface>
  [...]
  <Parameters>
    <Parameter Name="Remote Address"
               Id="param1"
               AccessFlags="!Tunable"
               DataType="Char[]"
               DefaultValue="192.168.1.22"
               RangeMin=""
               RangeMax=""
               Description="The IP of the receiving system."/>
    <Parameter Name="Remote Port"
               Id="param2"
               AccessFlags="!Tunable"
               DataType="UInt16"
               DefaultValue="44000"
               RangeMin="1"
               RangeMax="65535"
               Description="The port of the receiving system."/>
    <EnumerationParameter Name="Protocol"
                          Id="param3"
                          AccessFlags="!Tunable"
                          DataType="UInt32"
                          DefaultIndex="1"
                          Description="Protocol to be used. (UDP (default) or TCP)">
      <EnumerationValues>
        <EnumerationValue InternalValue="0"
                          DisplayName="UDP"
                          TraceValue="0" />
        <EnumerationValue InternalValue="1"
                          DisplayName="TCP"
                          TraceValue="1" />
      </EnumerationValues>
    </EnumerationParameter>
  </Parameters>
</ModelInterface>
```

Several properties are defined in the example:

Property	Description
`Remote_Address`	The IP of the receiving system.
`Remote_Port`	The port of the receiving system.
`Protocol`	The protocol to be used: UDP (default) or TCP.

The Protocol property is set via a list in ConfigurationDesk (`EnumerationParameter`). The `EnumerationValue` entries each define an item on the list. These values are interpreted as strings and used as inline parameters in the generated code.

The following illustration shows the **Properties Browser** dialog with the specified configuration properties.



Name	Value
Ethernet Send (1)	
Identification	
Name	Ethernet Send (1)
Function block type	Ethernet Send
Specific	
Remote Address	192.168.1.22
Remote Port	44000
Protocol	UDP
Referencing Configuration	
Ethernet request	Ethernet, 192.168.1.21 [Ethernet S...
Format	
Info	
Ethernet request [Ethernet ...	
Configuration	
Local port	44000
Info	
Internet protocol	IPv4

Note

If you specify port numbers for the Remote_port property, it is recommended to use port numbers from the range of registered ports (see http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers). Use numbers in the range 10000 … 49150.
The following port numbers are preconfigured:
- Ethernet Send: 44000
- Ethernet Receive: 44001

To avoid conflicts, do not use port numbers in the range of dynamic ports.

Using configuration properties in the C++ source code

The following macro is defined to read the values of configuration properties in the C++ source code:

```
Ethernet_Send_PARAMETER(<ParameterName>)
```

Related topics

Basics

References

Creating and Adjusting the Source Code Files (Ethernet)

General information

- For basic information on the source code files of custom function blocks, refer to Creating and Adjusting the Source Code Files on page 29.
- For a complete reference of the relevant XML elements in custom function blocks, refer to Hook Function Definitions on page 114.

Configuring the required network interface

To use the second interface in an Ethernet custom function block, you have to specify an IP address and a subnet mask.

The requested IP address and the subnet mask are configured in the creation function of the C++ source code. Example from `Ethernet_send.CPP`:

```
void ucf_Ethernet_Send_Create(DsTErrorList ErrorList, ucf_Ethernet_SendStruct_T * pBlockInstance)
{
    Ethernet_Send_INSTANCE_BUFFER_CREATE(pUD, Ethernet);

    char   LocalAddress [16];  // size the char array large enough to hold 4 x 3 digit numbers + 3 x dot + null
terminator
    UInt8 lenLA  = (pBlockInstance->Ethernet.Ethernet_request.Len_localIpAddress > 15) ? 15 : pBlockInstance-
>Ethernet.Ethernet_request.Len_localIpAddress;
    strncpy (LocalAddress, pBlockInstance->Ethernet.Ethernet_request.localIpAddress, lenLA);
    LocalAddress [lenLA] = '\0';

    char   RemoteAddress [16];
    UInt8 lenRA = (Ethernet_Send_PARAMETER(Len_Remote_Address) > 15) ? 15 : Ethernet_Send_PARAMETER(Len_Remote_Address);
    strncpy (RemoteAddress, Ethernet_Send_PARAMETER(Remote_Address), lenRA);
    RemoteAddress [lenRA] = '\0';

    /* Initialize user data */
    pUD->SendSize = sizeof(UInt32);
    pUD->Protocol = Ethernet_Send_PARAMETER(Protocol);

    memset(&pUD->LocalAddress, 0x0, sizeof(pUD->LocalAddress));
    pUD->LocalAddress.sin_port = htons(pBlockInstance->Ethernet.Ethernet_request.localPort);
    if (pBlockInstance->Ethernet.Ethernet_request.internetProtocolVersion == 1)
    {
        /* IPv4 interface assigned */
        pUD->LocalAddress.sin_family = AF_INET;
        int statusLA = inet_aton(LocalAddress, &pUD->LocalAddress.sin_addr);
        if (statusLA != 1)
        {
            printf("inet_aton() failed. <%s> is not a valid network address. ", LocalAddress);
            return;
        }
    }
    else
    {
        printf("Internet protocol (IP) version is not implemented.");
        return;
    }

    memset(&pUD->RemoteAddress, 0x0, sizeof(pUD->RemoteAddress));
    pUD->RemoteAddress.sin_family = AF_INET;
    pUD->RemoteAddress.sin_port   = htons(Ethernet_Send_PARAMETER(Remote_Port));
    int statusRA                  = inet_aton(RemoteAddress, &pUD->RemoteAddress.sin_addr);
    if (statusRA != 1)
    {
        printf("inet_aton() failed. <%s> is not a valid network address. ",
RemoteAddress);
        return;
    }


    /* User log of current communication setup */
    printf("Ethernet_Send: %s Protocol %s, Remote Address %s, Remote Port: %u\n",
            "Create sending socket on ",
            Ethernet_Send_PARAMETER(Protocol) == 0 ? "UDP" : "TCP",
            RemoteAddress,
            Ethernet_Send_PARAMETER(Remote_Port));

}
```

Related topics

Basics

Basics on Function Triggers (ConfigurationDesk Real-Time Implementation Guide 📖)

References

Specifying the Required Network Interface

Introduction

The Ethernet custom function blocks require a network interface in order to be executed and perform the I/O functionality.

Defining an Ethernet request in the XML file for Ethernet_Send

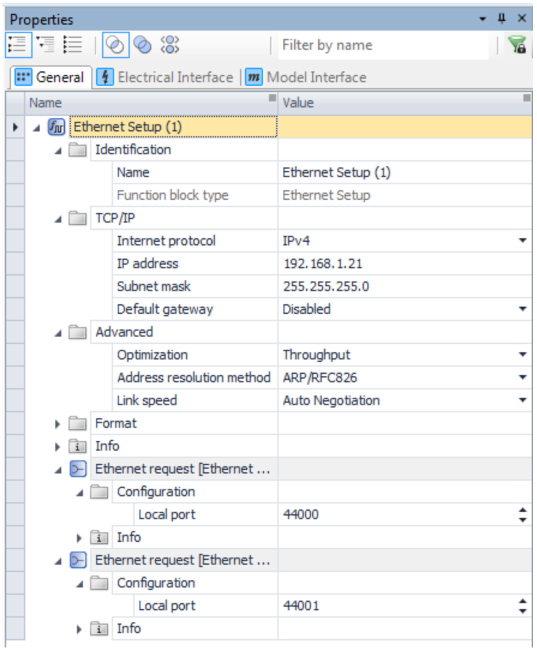
An Ethernet request can be specified for the `CustomFunctionBlock` element in the XML file as a `Feature` that is part of the `PhyscialLayerInterface` element. Example from `Ethernet_Send.xml`:

```
<PhysicalLayerInterfaces>
  <PhysicalLayerInterface Name="Ethernet" Id="sc1">
    <Features>
      <Feature Name="Ethernet request" Id="Feat1" FeatureName="EthernetRequest" >
        <MultiplicityConstraint MinNoOfConnections="1"/>
      </Feature>
    </Features>
  </PhysicalLayerInterface>
  <AdditionalWirings />
</PhysicalLayerInterfaces>
```

The `MultiplicityConstraint` element defines that a connection to the Ethernet request is required. Otherwise, a Function block: Missing or supernumerous block connections (by constraint) conflict is caused.

Specifying the required network interface via Properties Browser

The network interface that is required by the custom function block is specified via the Properties Browser for the Ethernet Send and the Ethernet Receive custom function block. Assign an Ethernet Setup or Virtual Ethernet Setup block via the Ethernet request property. Via the Ethernet request subelement you can define the Local port for sending and receiving. After you assign the same setup block to **Ethernet Send** and **Ethernet Receive**, you can configure both **Local port** properties at the setup block as well.



Name	Value
Ethernet Setup (1)	
Identification	
Name	Ethernet Setup (1)
Function block type	Ethernet Setup
TCP/IP	
Internet protocol	IPv4
IP address	192.168.1.21
Subnet mask	255.255.255.0
Default gateway	Disabled
Advanced	
Optimization	Throughput
Address resolution method	ARP/RFC826
Link speed	Auto Negotiation
Format	
Info	
Ethernet request [Ethernet ...	
Configuration	
Local port	44000
Info	
Ethernet request [Ethernet ...	
Configuration	
Local port	44001
Info	

Note

The Ethernet demo custom function blocks support only IPv4. Configuring IPv6 at the **Ethernet Setup** function block causes an error message when executing the real-time application.

Limitations for Implementing Custom Function Blocks

Objective

There are some limitations for implementing custom function blocks.

Limitations for Naming

Avoiding Simulink model names

Simulink models used in the ConfigurationDesk application must not have the same name as any of the custom function files in the custom function directories. For example, you must not use `UART_RS232_FlowControl.mdl` as a model name, because `UART RS232 FlowControl` is the name of a custom function.

Custom Function Block XML Element Reference

Schema version

The XML element reference refers to the current schema version `CustomFunctionTypeSchema_V3_1.xsd`.

Where to go from here

Information in this section

Introduction to the Custom Function Block XML Element Reference

Where to go from here

Information in this section

Basics on the Custom Function XML Schema

XML schema

An XML schema describes the structure of an XML document. It defines the elements and attributes to be used in the XML file. It also defines the hierarchical structure and the multiplicity of the elements. You can find the current and previous schema versions for ConfigurationDesk custom function XML files here:

```
<InstallationFolder>\ConfigurationDesk\Implementation\UserFiles
```

With ConfigurationDesk Version 5.3, a new XML schema was defined for custom function blocks (`CustomFunctionTypeSchema_V3_1.xsd`). ConfigurationDesk lets you update custom function files from the last schema version (`CustomFunctionTypeSchema_V3`, introduced with ConfigurationDesk Version 4.3) to the new schema via the **Create Updated Custom Function Type XML** (ConfigurationDesk User Interface Reference 📖) command.

Note

- Custom function blocks created according to previous schema versions are still available in your ConfigurationDesk application.
- You cannot downgrade an updated custom function file to the previous schema version. You are recommended to back up the existing files before the update.
- To have access to the newest features, you are recommended to create new custom function XML files according to the current XML schema version. To accesss these features in an existing custom function block, you have to update it.

Using the XML Element Reference

Construction kit for building custom function XML files

The XML element reference serves as a construction kit for building custom function XML files. Child elements, parent elements, and attributes are provided for each element. Every XML file starts at the root element. `CustomFunctionType` is the root element for a custom function XML file.

Documentation structure

For each XML element, the XML element reference is divided into the following sections:

Description Describes the purpose and characteristics of the XML element.

Parent elements The elements that the current element can be used in in the XML structure.

Example:

```
<MakeConfiguration>
  <Artefact Name="socket" Type="Library" />
</MakeConfiguration>
```

Here, the `MakeConfiguration` element is the parent element of the `Artefact` element (which is its child element) (refer to MakeConfiguration on page 120, Artefact on page 121).

Child elements The elements that can be used in the current element in the XML structure. The Multiplicity column provides the minimum and maximum number of occurrences in the custom function XML file.

Example:

```
<EnumerationValues>
  <EnumerationValue InternalValue="0" DisplayName="UDP" TraceValue="0" />
  <EnumerationValue InternalValue="1" DisplayName="TCP" TraceValue="1" />
</EnumerationValues>
```

Here, both `EnumerationValue` elements are child elements of the `EnumerationValues` element (which is their parent element). The multiplicity of the `EnumerationValue` element is 1..unbounded, which means that there must be at least one `EnumerationValue` element if an `EnumerationValues` element exists (refer to EnumerationValues on page 113, EnumerationValue on page 113).

Attributes The attributes that can be specified for the element. The Use column shows if the attribute is required and provides existing default values.

Example:

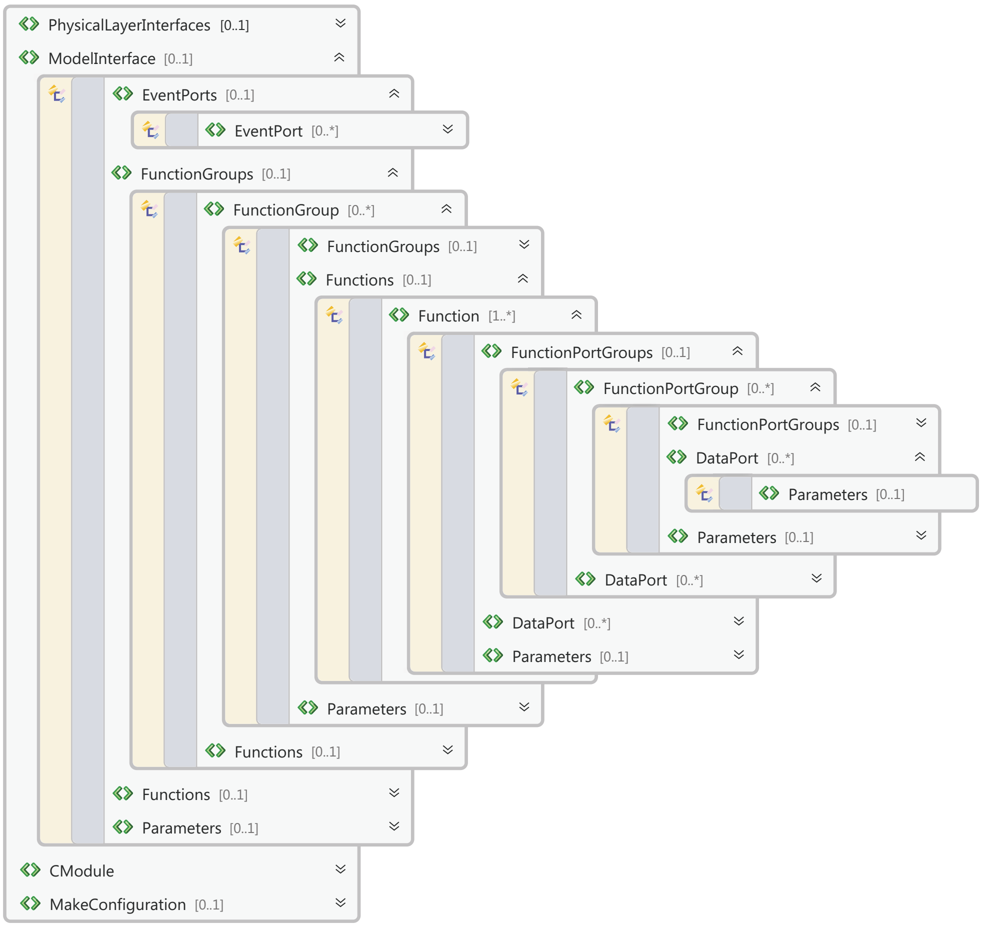
```
<Parameter Name="Interface Id" Id="param1" AccessFlags="!Tunable"
DataType="Char[]" DefaultValue="wm1" RangeMin="" RangeMax="" />
```

Several attributes are specified for the `Parameter` element. Some are required (e.g., `Name`), some are optional (e.g., `AccessFlags`) (refer to Parameter on page 109).

Navigating the XML hierarchy

You can navigate the XML hierarchy by clicking the linked child or parent elements.

For each of the main building blocks of the custom function XML file, the reference provides an overview of the descendant structure. For an example of a model interface overview, refer to the following illustration:



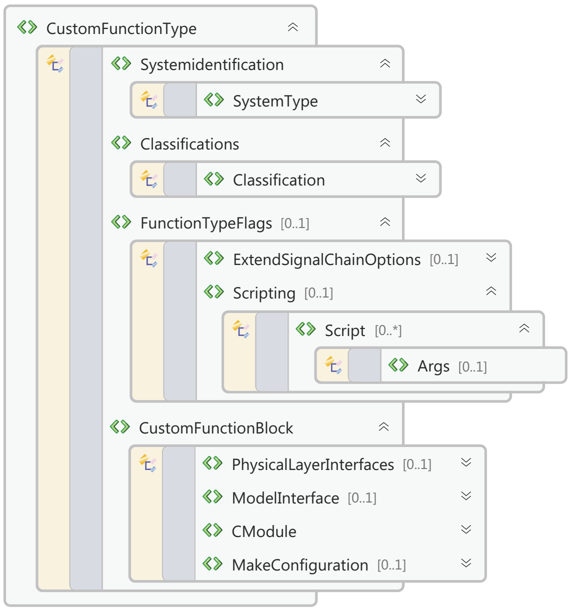
CustomFunctions

Where to go from here

Information in this section

Custom Function Type

Hierarchy overview



Where to go from here

Information in this section

CustomFunctionType

Description

Lets you define a function block type for a specified system.

Parent elements

None

Child elements

Name	Multiplicity	Purpose
Classifications on page 72	1..1	Hierarchy of classifications below the default 'Custom Functions' classification
CustomFunctionBlock on page 76	1..1	Definition of function block type
FunctionTypeFlags on page 73	0..1	Flags to define type-specific behaviors
Systemidentification on page 71	1..1	Specification of the target system

Attributes

Name	Use	Description
Version	required	Schema version the XML description file is based on Fixed value: ▪ 3.1

Systemidentification

Description

Lets you specify which system the function block type is defined for.

Parent elements

Name	Purpose
CustomFunctionType on page 70	System-specific definition of a function block type

Child elements

Name	Multiplicity	Purpose
SystemType on page 71	1..1	Specification of the system type

Attributes

None

SystemType

Description

Lets you specify which system type the function block type is designed for.

Parent elements

Name	Purpose
Systemidentification on page 71	Specification of the target system

Child elements None

Attributes

Name	Use	Description
Name	required	Possible values: ▪ SCALEXIO ▪ MABX

Classifications

Description Lets you structure installed Custom Functions by building up a hierachy of classifications below the default 'Custom Functions' classification.

Parent elements

Name	Purpose
CustomFunctionType on page 70	System-specific definition of a function block type

Child elements

Name	Multiplicity	Purpose
Classification on page 72	1..10	Classification defining a hierarchy level

Attributes None

Classification

Description Lets you define a hierarchy level by providing a corresponding classification.

Parent elements

Name	Purpose
Classifications on page 72	Hierarchy of classifications below the default 'Custom Functions' classification

Child elements None

Attributes

Name	Use	Description
Name	required	Classification name

FunctionTypeFlags

Description Lets you define the type-specific behavior for various aspects.

For example, you can define how model port blocks will be generated via the 'Extend signal chain' operation.

Parent elements

Name	Purpose
CustomFunctionType on page 70	System-specific definition of a function block type

Child elements

Name	Multiplicity	Purpose
ExtendSignalChainOptions on page 74	0..1	Defines the behavior for the extend signal chain operation.
Scripting on page 74	0..1	Definition of scripts. These will be triggerable on each block instance via its context menu (in a working view).

Attributes None

ExtendSignalChainOptions

Description

Lets you explicitly define the behavior for the 'Extend signal chain' operation.

If nothing is registered, the global configuration in ConfigurationDesk is applied.

Otherwise, the configuration will be overridden for the actual function block type, letting the type-specific options take effect.

Parent elements

Name	Purpose
FunctionTypeFlags on page 73	Flags to define type-specific behaviors

Child elements

None

Attributes

Name	Use	Description
PortDataTypes	optional	Defines the data type of the generated model ports. Possible values: ▪ Double: All generated model ports will have the data type Double/Float64. ▪ Inherited: All generated model ports will inherit the data type from the corresponding data ports.
BlockGenerationSchema	optional	Defines the schema for generating model port blocks. Possible values: ▪ Group model ports: The generated model ports will be grouped in (structured) model port blocks. ▪ One block per port: For each data port, a single model port block with a corresponding model port is generated.

Scripting

Description

Lets you define scripts that will be triggerable on each block instance via its context menu (in a working view).

Parent elements

Name	Purpose
FunctionTypeFlags on page 73	Flags to define type-specific behaviors

Child elements

Name	Multiplicity	Purpose
Script on page 75	1..unbounded	Definition of a script. This will be triggerable on each block instance via its context menu (in a working view).

Attributes None

Script

Description Lets you define a script that will be triggerable on each block instance via its context menu (in a working view).

Parent elements

Name	Purpose
Scripting on page 74	Definition of scripts. These will be triggerable on each block instance via its context menu (in a working view).

Child elements

Name	Multiplicity	Purpose
Args on page 76	0..1	A list of arguments passed to the script when being invoked. For example: <Args>--log -o "output.txt"</Args>.

Attributes

Name	Use	Description
Language	required	The script language Fixed value: ▪ Python

Name	Use	Description
Name	required	The name of the script file (including its file extension).
DisplayName	optional	Optional name used for displaying only (i.e. as name of the context menu entry). If not specified, the 'Name' attribute without the file extension is used.
Directory	optional	The directory the script file is located in. This attribute is mandatory if the script file is not located parallel to the Custom Function XML file. The value can be either a relative or an absolute path.

Args

Description

Lets you define a list of arguments passed to the script when being invoked. For example: <Args>--log -o "output.txt"</Args>.

Parent elements

Name	Purpose
Script on page 75	Definition of a script. This will be triggerable on each block instance via its context menu (in a working view).

Child elements

None

Attributes

None

CustomFunctionBlock

Description

Lets you define the function block type.

A function block type consists of several groups of information. For example, the model interface is mainly defined by functions, their function ports and user-defined configuration parameters. The code generation is defined by the user code module with its user access points.

Parent elements

Name	Purpose
CustomFunctionType on page 70	System-specific definition of a function block type

Child elements

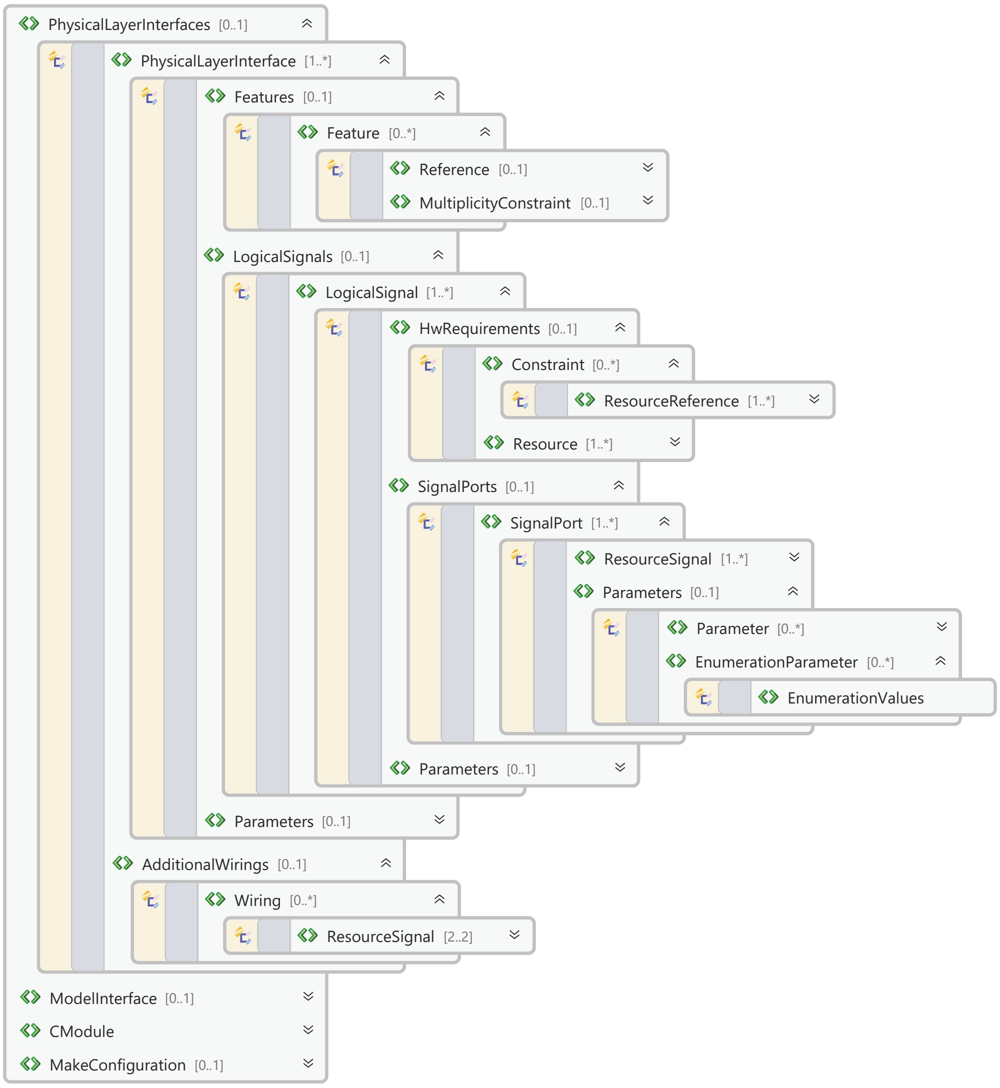
Name	Multiplicity	Purpose
CModule on page 115	1..1	Definition of the user code module
MakeConfiguration on page 120	0..1	Make configuration
ModelInterface on page 100	0..1	Definition of the model interface
PhysicalLayerInterfaces on page 79	0..1	Definition of physical layer interfaces

Attributes

Name	Use	Description
Version	required	The version information will be used for updates and migration.
Name	required	Defines the name of the Custom Function type.
DistributorInformation	optional	Lets you define some distributor information for the Custom Function type.
IsStopStatusOutputSupported	(default: false)	If set to 'true', the common 'Stop status output' parameter will be available. This lets you configure the 'Stop value' of a DataPort with the direction 'In'.
IsValueSaturationSupported	(default: false)	If set to 'true', the common 'Saturation usage' parameter will be available. This lets you configure the 'Saturation minimum/maximum value' of a DataPort mostly with the direction 'In'.

Electrical Interface

Hierarchy overview



Where to go from here

Information in this section

PhysicalLayerInterfaces

Description

Lets you define physical layer interfaces (electrical interface units) for the function block type.

Physical layer interfaces provide the interface of the function block to the external devices and to the real-time HW (via HW resource assignment). Each physical layer interface of a function block usually needs a different channel set to be assigned to. It also provides properties to configure the characteristics of the HW.

Parent elements

Name	Purpose
CustomFunctionBlock on page 76	Definition of function block type

Child elements

Name	Multiplicity	Purpose
AdditionalWirings on page 98	0..1	Definition of additional wiring connections
PhysicalLayerInterface on page 80	1..unbounded	Definition of a physical layer

Attributes

None

PhysicalLayerInterface

Description

Lets you define a physical layer interface (electrical interface unit) for the function block type.

A physical layer interface provides an interface of the function block to the external devices and to the real-time HW (via HW resource assignment). It also provides properties to configure the characteristics of the HW.

Parent elements

Name	Purpose
PhysicalLayerInterfaces on page 79	Definition of physical layer interfaces

Child elements

Name	Multiplicity	Purpose
Features on page 81	0..1	Definition of special features
LogicalSignals on page 84	0..1	Definition of logical signals
Parameters on page 108	0..1	Lets you define type-specific configuration parameters: e.g., simple configuration parameters or enumeration parameters.

Attributes

Name	Use	Description
Name	required	Name of the element
Id	required	The ID has to be unique within the description file. It will be used for referencing.

Name	Use	Description
ReadonlyChannelRequests	optional (default: false)	If set to 'true', the defined HW requirements (channel requests) are set to be read-only. All channels will be automatically assigned by selecting a resource provider (channel set). A manual assignment will not be possible.

Features

Description

Lets you define various special features, e.g., a request for connecting to an angular clock provider.

Parent elements

Name	Purpose
PhysicalLayerInterface on page 80	Definition of a physical layer

Child elements

Name	Multiplicity	Purpose
Feature on page 81	0..unbounded	Definition of a feature

Attributes

None

Feature

Description

Lets you define a feature, e.g., a request for connecting to an angular clock provider.

Parent elements

Name	Purpose
Features on page 81	Definition of special features

Child elements

Name	Multiplicity	Purpose
MultiplicityConstraint on page 83	0..1	Constraint regarding the connection multiplicity of the defined feature.
Reference on page 83	0..1	Referenced HW resource the defined feature is using.

Attributes

Name	Use	Description
Name	required	Name of the element
Id	required	The ID has to be unique within the description file. It will be used for referencing.
FeatureName	required	Definition of the feature type Possible values: ▪ ApuDependend: The function block depends on an angular clock that is indirectly provided by an APU-providing function block. ▪ ApuProvider: The function block provides a master APU. ▪ BlockProvider: Via a BlockProvider feature the function block will be referencable by other function blocks. It will be assignable at a requesting block if the requesting block requires the same role as being defined for this BlockProvider feature. ▪ BlockRequest: Via a BlockRequest feature the function block can reference another function block. A function block (with BlockProvider features) will be assignable here if it provides a role that is requested by this BlockRequest feature. ▪ EthernetRequest: Via an EthernetRequest feature the function block can reference an Ethernet Setup block.
AngleRange	optional	Specifies the mode for a defined angular clock (master or slave APU). In case of the 'ApuProvider' feature, this attribute is mandatory. You can specify multiple angle ranges to support different modes. In such a case, the corresponding angle range property will be configurable within the tool. The initial value will be the first specified mode.
Roles	optional	Specifies the roles for BlockProvider and BlockRequest features. In case of a BlockProvider just one role is allowed, for a BlockRequest it can be multiple.

Reference

Description

Lets you specify a HW resource that should be used for a defined feature, e.g. a master APU provider.

Parent elements

Name	Purpose
Feature on page 81	Definition of a feature

Child elements

None

Attributes

Name	Use	Description
Resource	required	ID of the HW resource to be referenced.

MultiplicityConstraint

Description

Lets you constrain the multiplicity of connections to or from the defined feature, e.g. a master APU request.

Parent elements

Name	Purpose
Feature on page 81	Definition of a feature

Child elements

None

Attributes

Name	Use	Description
MinNoOfConnections	optional	Minimum number of connections. For requesting features, only the values '0' and '1' are allowed.
MaxNoOfConnections	optional	Maximum number of connections (only for providing features). The value '0' is not allowed.

LogicalSignals

Description

Lets you define logical signals. Logical signals are used to group signal ports and relate to HW resources.

Parent elements

Name	Purpose
PhysicalLayerInterface on page 80	Definition of a physical layer

Child elements

Name	Multiplicity	Purpose
LogicalSignal on page 84	1..unbounded	Definition of a logical signal

Attributes

None

LogicalSignal

Description

Lets you define a logical signal. A logical signal is used to group signal ports and relates to HW resources.

Parent elements

Name	Purpose
LogicalSignals on page 84	Definition of logical signals

Child elements

Name	Multiplicity	Purpose
HwRequirements on page 85	0..1	Definition of HW resource requirements
Parameters on page 108	0..1	Lets you define type-specific configuration parameters: e.g., simple configuration parameters or enumeration parameters.
SignalPorts on page 91	0..1	Definition of signal ports

Attributes

Name	Use	Description
Name	required	Name of the element
Id	required	The ID has to be unique within the description file. It will be used for referencing.

HwRequirements

Description

Lets you define HW resource requirements. The requirements are defined as concrete channel requests (hwResourceType) that are structured in a tree. The Constraints apply to all channel requests in the current subgroup or below.

Parent elements

Name	Purpose
LogicalSignal on page 84	Definition of a logical signal

Child elements

Name	Multiplicity	Purpose
Constraint on page 86	0..unbounded	Definition of a channel constraint
Resource on page 87	1..unbounded	Definition of a channel request

Attributes

None

Constraint

Description

Lets you define a channel constraint.

Parent elements

Name	Purpose
HwRequirements on page 85	Definition of HW resource requirements

Child elements

Name	Multiplicity	Purpose
ResourceReference on page 86	1..unbounded	Referenced resource being affected by the constraint.

Attributes

Name	Use	Description
Name	required	Channel constraint type Possible values: ▪ ConsecutiveChannels: All channels that are affected by the constraint must be next to each other. ▪ SameBoard: All channels that are affected by the constraint must be on the same I/O board.

ResourceReference

Description

Lets you reference a resources to be affected by the constraint.

Parent elements

Name	Purpose
Constraint on page 86	Definition of a channel constraint

Child elements

None

Attributes

Name	Use	Description
ResourceId	required	The ID of a defined HW resource being affected by the constraint.

Resource

Description

Lets you define a channel request. A concrete channel request indicates which HW properties are required and what pins are expected at the assigned channel.

The name of the channel request will be displayed in configuration dialogs, e.g., the property grid.

Parent elements

Name	Purpose
HwRequirements on page 85	Definition of HW resource requirements

Child elements

None

Attributes

Name	Use	Description
Name	required	Name of the element
Id	required	The ID has to be unique within the description file. It will be used for referencing.
ResourceType	required	Resource type defining the assignable HW resources. Possible values: ▪ Acceleration_Sensor_Unit_1:

Name	Use	Description
		For future use only. ▪ Analog_In_1 ▪ Analog_In_2 ▪ Analog_In_4: This channel type does not support failure simulation or load functionality. ▪ Analog_In_5: This channel type does not support failure simulation or load functionality. ▪ Analog_In_6: This channel type does not support failure simulation or load functionality. ▪ Analog_In_16: This channel type does not support failure simulation or load functionality. ▪ Analog_In_17: This channel type does not support failure simulation or load functionality. ▪ Analog_Out_1 ▪ Analog_Out_2 ▪ Analog_Out_3 ▪ Analog_Out_4 ▪ Analog_Out_6: This channel type does not support failure simulation. ▪ Analog_Out_7: This channel type does not support failure simulation. ▪ Analog_Out_8: This channel type does not support failure simulation. ▪ Analog_Out_9: This channel type does not support failure simulation. ▪ Analog_Out_10: This channel type does not support failure simulation. ▪ Angle_Unit_Set: The angle unit set from a processing unit. This channel type does not support failure simulation or signal ports. ▪ Angle_Unit_Set_2: The angle unit set from an I/O board.

Name	Use	Description
		This channel type does not support failure simulation or signal ports. ▪ Bus_1 ▪ CAN_1 ▪ CAN_2 ▪ CAN_6 ▪ Digital_In_1 ▪ Digital_In_2 ▪ Digital_In_3: This channel type does not support failure simulation or load functionality. ▪ Digital_InOut_1 ▪ Digital_InOut_3: This channel type does not support failure simulation or load functionality. ▪ Digital_InOut_5: This channel type does not support failure simulation or load functionality. ▪ Digital_InOut_9: This channel type does not support failure simulation or load functionality. ▪ Digital_InOut_10: This channel type does not support failure simulation or load functionality. ▪ Digital_Out_1 ▪ Digital_Out_2 ▪ Digital_Out_3: This channel type does not support failure simulation. ▪ Digital_Out_8: This channel type does not support failure simulation. ▪ DS2655 ▪ DS2656 ▪ DS6601 ▪ DS6602 ▪ Ethernet: This channel type does not support failure simulation or signal ports. ▪ Flexible_In_1 ▪ Flexible_In_2 ▪ Flexible_In_3: This channel type does not support failure simulation or load functionality. ▪ Flexible_InOut_1: This channel type does not support failure simulation or load functionality. ▪ Flexible_Out_1 ▪ FlexRay_1 ▪ FlexRay_2 ▪ FlexRay_4 ▪ LED_Out_1 ▪ LIN_1 ▪ LIN_2 ▪ LIN_4 ▪ Load_1 ▪ Power_Control_1: This channel type does not support failure simulation or signal ports. ▪ Power_Switch_1: This channel type does not support failure simulation. ▪ Power_Switch_2: This channel type does not support failure simulation. ▪ Pressure_Sensor_Unit_1:

Name	Use	Description
		For future use only. ▪ Resistance_Out_1 ▪ Resistance_Out_2: This channel type does not support failure simulation. ▪ Resolver_In_2 ▪ Trigger_In_1: This channel type does not support failure simulation or load functionality. ▪ Trigger_In_2: This channel type does not support failure simulation or load functionality. ▪ UART_1 ▪ UART_4 ▪ UART_5 ▪ UART_6 ▪ IMBus_Analog_In_7 ▪ IMBus_Analog_In_8 ▪ IMBus_Analog_In_9 ▪ IMBus_Analog_In_10 ▪ IMBus_Analog_In_11 ▪ IMBus_Analog_In_12 ▪ IMBus_Analog_In_13 ▪ IMBus_Analog_In_14 ▪ IMBus_Analog_In_15 ▪ IMBus_Analog_Out_11 ▪ IMBus_Analog_Out_12 ▪ IMBus_Analog_Out_13 ▪ IMBus_Analog_Out_14 ▪ IMBus_CAN_3 ▪ IMBus_CAN_4 ▪ IMBus_CAN_5 ▪ IMBus_Digital_In_4 ▪ IMBus_Digital_In_5 ▪ IMBus_Digital_In_6 ▪ IMBus_Digital_In_7 ▪ IMBus_Digital_In_8 ▪ IMBus_Digital_In_9 ▪ IMBus_Digital_In_10 ▪ IMBus_Digital_InOut_6 ▪ IMBus_Digital_InOut_7 ▪ IMBus_Digital_InOut_8 ▪ IMBus_Digital_Out_4 ▪ IMBus_Digital_Out_5 ▪ IMBus_Digital_Out_6 ▪ IMBus_Digital_Out_7 ▪ IMBus_DS1514 ▪ IMBus_FlexRay_3 ▪ IMBus_Lambda_In_1 ▪ IMBus_LIN_3 ▪ IMBus_Resolver_In_1 ▪ IMBus_Trigger_In_3 ▪ IMBus_UART_2 ▪ IMBus_UART_3
ChannelName	optional	In the case of the resource type DS1514, DS2655, DS2656 and DS6601, the channel name attribute is mandatory.

SignalPorts

Description

Lets you define signal ports. Signal ports provide the interface to external devices (for example ECUs) via device blocks. They represent the electrical connection points of the function block.

Parent elements

Name	Purpose
LogicalSignal on page 84	Definition of a logical signal

Child elements

Name	Multiplicity	Purpose
SignalPort on page 91	1..unbounded	Definition of a signal port

Attributes

None

SignalPort

Description

Lets you define a signal port. A signal port provides an interface to external devices (for example ECUs) via device blocks. It represents an electrical connection point of the function block.

A signal port must only use ResourceSignals of HW resources that are requested by its LogicalSignal.

Parent elements

Name	Purpose
SignalPorts on page 91	Definition of signal ports

Child elements

Name	Multiplicity	Purpose
Parameters on page 108	0..1	Lets you define type-specific configuration parameters: e.g., simple configuration parameters or enumeration parameters.
ResourceSignal on page 92	1..unbounded	Definition of a ResourceSignal reference

Attributes

Name	Use	Description
Name	required	Name of the element
Id	required	The ID has to be unique within the description file. It will be used for referencing.
Direction	required	The port direction. Signal ports with the role 'Reference' must also specify the direction 'Reference'. Possible values: ▪ In: Measuring I/O functionality. ▪ Out: Generating I/O functionality. ▪ Bidirectional: Generating and measuring I/O functionality (mostly bus functionality). ▪ Reference
IsFiuEnabled	required	'True': failure simulation functionality (FIU) will be activated at the corresponding SignalPort. 'False': FIU will be deactivated at the corresponding SignalPort. Note: Not all HW resources or pin types support FIU.
IsLoadEnabled	(default: true)	'True': load functionality will be activated at the corresponding SignalPort. 'False': load functionality will be deactivated at the corresponding SignalPort. Note: Not all HW resources support load functionality.

ResourceSignal

Description

Lets you define a reference to a pin of a defined HW resource.

Parent elements

Name	Purpose
Wiring on page 98	Definition of a wiring connection
SignalPort on page 91	Definition of a signal port

Child elements

None

Attributes

Name	Use	Description
Name	required	Used pin of the referenced HW resource. Possible values: ▪ Signal: The Signal pin (of all channels of types Flexible_In_1, Flexible_In_2, Digital_In_1, Analog_In_1, Analog_In_2, Flexible_Out_1, Digital_Out_1, Analog_Out_1, Analog_Out_2, Analog_Out_3, Analog_Out_4, Resistance_Out_1, Power_Switch_1, Power_Switch_2, Load_1, Digital_In_2, Digital_Out_2, Digital_InOut_1) supports FIU and is protected by fuses. ▪ Reference: The Reference pin (of all channels of types Flexible_In_1, Analog_In_1, Analog_In_2, Flexible_Out_1, Digital_Out_1, Analog_Out_1, Analog_Out_2, Analog_Out_3, Analog_Out_4, Resistance_Out_1, Power_Switch_1, Power_Switch_2, CAN_1, CAN_2, LIN_1, LIN_2, FlexRay_1 and Load_1) is the reference potential of the Signal pin. All channels of types Flexible_In_2 and Digital_In_1 have a common reference pin. This Reference pin is a common and single reference potential of all Signal pins. As a best practice, it is recommended to define a common signal port (GND) which connects all Reference pins of all channels. ▪ PosLoad: To connect an external load to channels of types Flexible_In_1, Flexible_In_2, Digital_In_1 and Analog_In_1. To connect an external load (if a cable bridge is equipped) or an reference for the internal load (if an internal load is equipped) to channels of types Digital_In_2. ▪ NegLoad: To connect an external load to channels of type Flexible_In_1. ▪ ReferenceHigh: The ReferenceHigh pin of channels of type Digital_Out_1, Digital_Out_2 and Digital_InOut_1 (output mode) is the high reference potential of the Signal pin. To connect an external load (if a cable bridge is equipped) or an reference for the internal load (if an internal load is equipped) to channels of types Digital_InOut_1 (input mode). ▪ ReferenceHigh2: The ReferenceHigh2 pin of channels of type Digital_Out_1 and Digital_InOut_3 is the second high reference potential of the Signal pin. ▪ PinA: Multi-purpose pin of a channel of types Bus_1 and LIN_1. The specific purpose depends on the I/O functionality. The CAN-High pin of a channel of types CAN_1, CAN_2, CAN_6, IMBus_CAN_3, IMBus_CAN_4 and IMBus_CAN_5. The LIN signal pin of a channel of types LIN_2, LIN_4 and IMBus_LIN_3. The FlexRay-High pin of a channel of type FlexRay_1.

Name	Use	Description
		The serial RX-High pin of a channel of type IMBus_UART_3. The RE pin of a channel of type IMBus_Lambda_In_1. ▪ PinB: Multi-purpose pin of a channel of type Bus_1. The specific purpose depends on the I/O functionality. The CAN-Low pin of a channel of types CAN_1, CAN_2, CAN_6, IMBus_CAN_3, IMBus_CAN_4 and IMBus_CAN_5. The FlexRay-Low pin of a channel of type FlexRay_1. The serial RX-Low pin of a channel of type IMBus_UART_3. The IPE pin of a channel of type IMBus_Lambda_In_1. ▪ PinC: Multi-purpose pin of a channel of type Bus_1. The specific purpose depends on the I/O functionality. The CAN-High feedthrough pin of a channel of types CAN_2, CAN_6 and IMBus_CAN_5. The serial TX-High pin of a channel of type IMBus_UART_3. The APE pin of a channel of type IMBus_Lambda_In_1. ▪ PinD: Multi-purpose pin of a channel of type Bus_1. The specific purpose depends on the I/O functionality. The CAN-Low feedthrough pin of a channel of types CAN_2, CAN_6 and IMBus_CAN_5.

Name	Use	Description
		The serial TX-Low pin of a channel of type IMBus_UART_3.

Name	Use	Description
		The MES pin of a channel of type IMBus_Lambda_In_1. ▪ VBat: VBat pin of a channel of types CAN_1, CAN_2, LIN_1, LIN_2, FlexRay_1 and UART_1. ▪ SignalInternal: The internal Signal pin (of all channels of types Analog_In_2, Analog_Out_2) is for wiring (no FIU and fuses are available). ▪ ReferenceInternal: The internal Reference pin (of all channels of types Analog_In_2, Analog_Out_2) is for wiring (no FIU and fuses are available). ▪ FlexRayAHigh: The FlexRay-High pin of a channel of type FlexRay_2 for channel A communication. ▪ FlexRayALow: The FlexRay-Low pin of a channel of type FlexRay_2 for channel A communication. ▪ FlexRayAHighFT: The FlexRay-High feedthrough pin of a channel of type FlexRay_2 for channel A communication. ▪ FlexRayALowFT: The FlexRay-Low feedthrough pin of a channel of type FlexRay_2 for channel A communication. ▪ FlexRayAVbat: The VBat pin of a channel of type FlexRay_2 for channel A communication. ▪ FlexRayAGnd: The Reference pin of a channel of type FlexRay_2 for channel A communication. ▪ FlexRayBHigh: The FlexRay-High pin of a channel of type FlexRay_2 for channel B communication. ▪ FlexRayBLow: The FlexRay-Low pin of a channel of type FlexRay_2 for channel B communication. ▪ FlexRayBHighFT: The FlexRay-High feedthrough pin of a channel of type FlexRay_2 for channel B communication. ▪ FlexRayBLowFT: The FlexRay-Low feedthrough pin of a channel of type FlexRay_2 for channel B communication. ▪ FlexRayBVbat: The VBat pin of a channel of type FlexRay_2 for channel B communication. ▪ FlexRayBGnd: The Reference pin of a channel of type FlexRay_2 for channel B communication. ▪ FlexRayWakeup: The FlexRay-Wakeup pin as connection to an external wakeup-signal. ▪ Inhibit1 ▪ Inhibit2 ▪ ResolverExcSignal:

Name	Use	Description
		The Exc signal pin of a channel of type Resolver_1. ▪ ResolverSinSignal: The Sin signal pin of a channel of type Resolver_1. ▪ ResolverCosSignal: The Cos signal pin of a channel of type Resolver_1. ▪ ResolverExcReference: The Exc reference pin of a channel of type Resolver_1. ▪ ResolverSinReference: The Sin reference pin of a channel of type Resolver_1. ▪ ResolverCosReference: The Cos reference pin of a channel of type Resolver_1. ▪ UartCts: The CTS pin of a channel of types UART_5, UART_6. The CTS (high) pin of a channel of type UART_1. ▪ UartCtsLow: The CTS (low) pin of a channel of type UART_1. ▪ UartDcd: The DCD pin of a channel of types UART_5, UART_6. ▪ UartDsr: The DSR pin of a channel of types UART_5, UART_6. ▪ UartDtr: The DTR pin of a channel of types UART_5, UART_6. ▪ UartGnd: The signal ground pin of a channel of types UART_1, UART_4, UART_5, UART_6, IMBus_UART_2. ▪ UartRts: The RTS/RTR pin of a channel of types UART_5, UART_6. The RTS/RTR (high) pin of a channel of type UART_1. ▪ UartRtsLow: The RTS/RTR (low) pin of a channel of type UART_1. ▪ UartRx: The RX pin of a channel of types UART_5, UART_6, IMBus_UART_2. The RX (high) pin of a channel of types UART_1, UART_4. ▪ UartRxLow: The RX (low) pin of a channel of types UART_1, UART_4. ▪ UartTx: The TX pin of a channel of types UART_5, UART_6, IMBus_UART_2. The TX (high) pin of a channel of types UART_1, UART_4. ▪ UartTxLow: The TX (low) pin of a channel of types UART_1, UART_4. ▪ KLine: The K Line pin of a channel of type UART_1. ▪ LLine: The L Line pin of a channel of type UART_1.
ResourceId	required	The ID of a defined HW resource being used by this ResourceSignal.

AdditionalWirings

Description

Lets you define additional wiring connections.

Parent elements

Name	Purpose
PhysicalLayerInterfaces on page 79	Definition of physical layer interfaces

Child elements

Name	Multiplicity	Purpose
Wiring on page 98	0..unbounded	Definition of a wiring connection

Attributes

None

Wiring

Description

Lets you define a wiring connection consisting of a pair of ResourceSignals that should be wired at the external simulator connector panel.

Parent elements

Name	Purpose
AdditionalWirings on page 98	Definition of additional wiring connections

Child elements

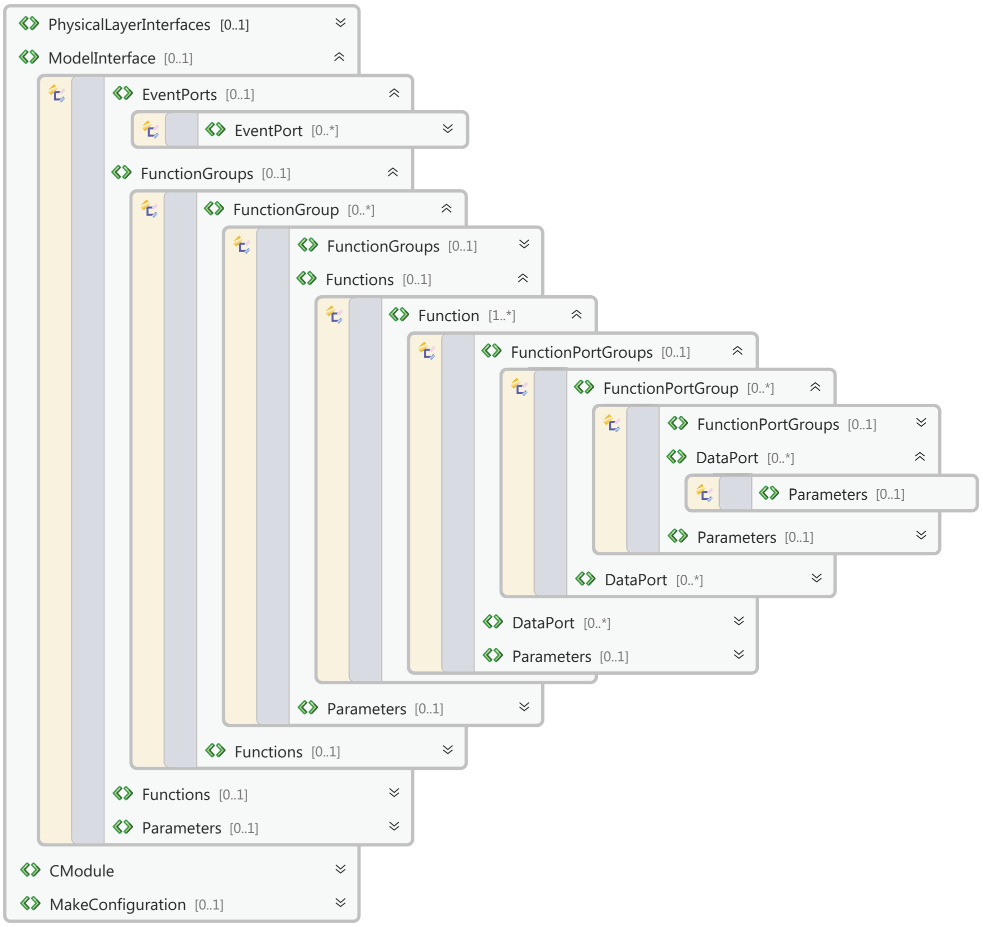
Name	Multiplicity	Purpose
ResourceSignal on page 92	2..2	Definition of the ResourceSignal reference

Attributes

None

Model Interface

Hierarchy overview



Where to go from here

Information in this section

ModelInterface

Description

Lets you define the model interface for the function block type.

Parent elements

Name	Purpose
CustomFunctionBlock on page 76	Definition of function block type

Child elements

Name	Multiplicity	Purpose
EventPorts on page 101	0..1	Definition of event ports
FunctionGroups on page 102	0..1	Lets you define function groups.
Functions on page 103	0..1	Lets you define functions providing the interface to the behavior model via model port blocks.
Parameters on page 108	0..1	Lets you define type-specific configuration parameters: e.g., simple configuration parameters or enumeration parameters.

Attributes

None

EventPorts

Description

Lets you define event ports.

Parent elements

Name	Purpose
ModelInterface on page 100	Definition of the model interface

Child elements

Name	Multiplicity	Purpose
EventPort on page 101	0..unbounded	Definition of an event port

Attributes

None

EventPort

Description

Lets you define an event port.

Parent elements

Name	Purpose
EventPorts on page 101	Definition of event ports

Child elements

None

Attributes

Name	Use	Description
Name	required	Name of the element
Id	required	The ID has to be unique within the description file. It will be used for referencing.

Name	Use	Description
EventIdentifier	required	The numeric identifier of the event if the I/O driver invokes several different events. The value is defined in the specific driver API. Set to 0 if the attached driver has only 1 event.
EventType	required	Currently not evaluated. Possible values: ▪ HW: Hardware event ▪ SW: Software event

FunctionGroups

Description

Lets you define function groups.

Parent elements

Name	Purpose
ModelInterface on page 100	Definition of the model interface
FunctionGroup on page 102	Lets you define a function group. A function group is used to group further function groups and/or functions.

Child elements

Name	Multiplicity	Purpose
FunctionGroup on page 102	0..unbounded	Lets you define a function group. A function group is used to group further function groups and/or functions.
Functions on page 103	0..1	Lets you define functions providing the interface to the behavior model via model port blocks.

Attributes

None

FunctionGroup

Description

Lets you define a function group. A function group is used to group further function groups and/or functions.

Parent elements

Name	Purpose
FunctionGroups on page 102	Lets you define function groups.

Child elements

Name	Multiplicity	Purpose
FunctionGroups on page 102	0..1	Lets you define function groups.
Functions on page 103	0..1	Lets you define functions providing the interface to the behavior model via model port blocks.
Parameters on page 108	0..1	Lets you define type-specific configuration parameters: e.g., simple configuration parameters or enumeration parameters.

Attributes

Name	Use	Description
Name	required	Name of the element
Id	required	The ID has to be unique within the description file. It will be used for referencing.

Functions

Description

Lets you define functions providing the interface to the behavior model via model port blocks.

Parent elements

Name	Purpose
ModelInterface on page 100	Definition of the model interface
FunctionGroups on page 102	Lets you define function groups.
FunctionGroup on page 102	Lets you define a function group. A function group is used to group further function groups and/or functions.

Child elements

Name	Multiplicity	Purpose
Function on page 104	1..unbounded	Lets you define a function providing an interface to the behavior model via model port blocks. A function can contain one or more data function ports (function signals) that are triggered together. The user can specify whether multiple triggers are allowed or not.

Attributes

None

Function

Description

Lets you define a function providing an interface to the behavior model via model port blocks.

A function can contain one or more data function ports (function signals) that are triggered together. The user can specify whether multiple triggers are allowed or not.

Parent elements

Name	Purpose
Functions on page 103	Lets you define functions providing the interface to the behavior model via model port blocks.

Child elements

Name	Multiplicity	Purpose
DataPort on page 106	0..unbounded	Lets you define a data port.
FunctionPortGroups on page 105	0..1	Lets you define function port groups.
Parameters on page 108	0..1	Lets you define type-specific configuration parameters: e.g., simple configuration parameters or enumeration parameters.

Attributes

Name	Use	Description
Name	required	Name of the element
Id	required	The ID has to be unique within the description file. It will be used for referencing.
IsMultipleTriggerable	(default: false)	Defines if the function is allowed to be triggered multiple times (e.g., in different timer tasks).

FunctionPortGroups

Description Lets you define function port groups.

Parent elements

Name	Purpose
FunctionPortGroup on page 106	Lets you define a function port group. A function port group is used to group further function port groups and/or data ports.
Function on page 104	Lets you define a function providing an interface to the behavior model via model port blocks. A function can contain one or more data function ports (function signals) that are triggered together. The user can specify whether multiple triggers are allowed or not.

Child elements

Name	Multiplicity	Purpose
DataPort on page 106	0..unbounded	Lets you define a data port.
FunctionPortGroup on page 106	0..unbounded	Lets you define a function port group. A function port group is used to group further function port groups and/or data ports.

Attributes None

FunctionPortGroup

Description

Lets you define a function port group. A function port group is used to group further function port groups and/or data ports.

Parent elements

Name	Purpose
FunctionPortGroups on page 105	Lets you define function port groups.

Child elements

Name	Multiplicity	Purpose
DataPort on page 106	0..unbounded	Lets you define a data port.
FunctionPortGroups on page 105	0..1	Lets you define function port groups.
Parameters on page 108	0..1	Lets you define type-specific configuration parameters: e.g., simple configuration parameters or enumeration parameters.

Attributes

Name	Use	Description
Name	required	Name of the element
Id	required	The ID has to be unique within the description file. It will be used for referencing.

DataPort

Description

Lets you define a data port.

Parent elements

Name	Purpose
FunctionPortGroups on page 105	Lets you define function port groups.
FunctionPortGroup on page 106	Lets you define a function port group. A function port group is used to group further function port groups and/or data ports.

Name	Purpose
Function on page 104	Lets you define a function providing an interface to the behavior model via model port blocks. A function can contain one or more data function ports (function signals) that are triggered together. The user can specify whether multiple triggers are allowed or not.

Child elements

Name	Multiplicity	Purpose
Parameters on page 108	0..1	Lets you define type-specific configuration parameters: e.g., simple configuration parameters or enumeration parameters.

Attributes

Name	Use	Description
Name	required	Name of the element
Id	required	The ID has to be unique within the description file. It will be used for referencing.
Unit	required	-
Width	required	In the case of Width > 1, all elements of the vectorial function signal have identical properties such as RangeMax, Unit, ...
MaxWidth	optional	If specified, the port width becomes adjustable in the range of [1..MaxWidth].
Direction	required	The port view direction of the function signal. Possible values: ▪ In: The function signal goes from the model to the I/O driver. ▪ Out: The function signal goes from the I/O driver to the model.
RangeMin	optional	If missing, the range of the given 'DataType' will be assumed.
RangeMax	optional	If missing, the range of the given 'DataType' will be assumed.
DefaultValue	required	You can define a vectorial parameter by separating single values by ';'. Note: All values must have the same data type.
DataType	required	Possible values: ▪ Bool ▪ UInt8 ▪ UInt16 ▪ UInt32 ▪ UInt64 ▪ Int8 ▪ Int16 ▪ Int32 ▪ Int64 ▪ Float32

Name	Use	Description
		▪ Float64 ▪ dsfloat
PortType	optional	In general, the FunctionPort type (scalar or vectorial) is implicitly determined using the specified data width. Via this parameter, this can be done explicitly, offering some additional options. Possible values: ▪ Scalar: A scalar function port. ▪ Vector: A standard vectorial vector port. ▪ CondensedVector: A special vectorial port in ConfigurationDesk where the port values (e.g., 'Initial value') always have a data width of 1 defining the common vector element value of the underlying property.

Properties

Where to go from here

Information in this section

Parameters

Description

Lets you define type-specific configuration parameters: e.g., simple configuration parameters or enumeration parameters.

Parent elements

Name	Purpose
ModelInterface on page 100	Definition of the model interface
PhysicalLayerInterface on page 80	Definition of a physical layer
LogicalSignal on page 84	Definition of a logical signal
FunctionGroup on page 102	Lets you define a function group. A function group is used to group further function groups and/or functions.
FunctionPortGroup on page 106	Lets you define a function port group. A function port group is used to group further function port groups and/or data ports.
SignalPort on page 91	Definition of a signal port
Function on page 104	Lets you define a function providing an interface to the behavior model via model port blocks. A function can contain one or more data function ports (function signals) that are triggered together. The user can specify whether multiple triggers are allowed or not.
DataPort on page 106	Lets you define a data port.

Child elements

Name	Multiplicity	Purpose
EnumerationParameter on page 111	0..unbounded	Definition of an enumeration parameter
Parameter on page 109	0..unbounded	Definition of a parameter

Attributes

None

Parameter

Description

Lets you define a configuration parameter. A configuration paramter is defined by its name, data type and initial value.

The configured value can be accessed in the user code via the instance-specific runtime struct.

Parent elements

Name	Purpose
Parameters on page 108	Lets you define type-specific configuration parameters: e.g., simple configuration parameters or enumeration parameters.

Child elements

None

Attributes

Name	Use	Description
Name	required	Name of the element
Id	required	The ID has to be unique within the description file. It will be used for referencing.
DisplayName	optional	Name used for displaying only (e.g., in the property grid).
Description	optional	Documentation of the parameter. Will be displayed in property grid.
Category	optional	Optional, specific parent category of the parameter. Will be displayed in property grid.
AccessFlags	optional	Defines how and in which contexts the parameter can be accessed (default: editable, tunable, visible).
TraceType	optional	Defines the type of the tunable parameter within the experimentation tool (default: configuration parameter). Possible values: ▪ Configuration: The parameter is a configuration parameter. ▪ Measurement: The parameter is a measurement parameter.

SubstructureForVDF	optional	Defines an optional substructure for a tunable parameter within the variable description file.
		If not specified or empty, the parameter will be added at root level (root => parent element the parameter is generated at).
		Example: "subnode1:subsubnodeA" results in
		group "parent element"
		group "subnode1"
		group "subsubnodeA"
		tunable_Parameter
		{
		type: [...]
		alias: [...]
		flags: [...]
		range: [...]
		desc: [...]
		}
		endgroup

Name	Use	Description
		endgroup endgroup
DataType	required	In the case of data type char[], the attributes 'RangeMin' and 'RangeMax' will be ignored. char[] parameters will not be included in the TRC file.
DefaultValue	required	You can define a vectorial parameter by separating single values by ';'. All values must have the same data type.
RangeMin	optional	If missing, the range of the given 'DataType' will be assumed.
RangeMax	optional	If missing, the range of the given 'DataType' will be assumed.
Unit	optional	-

EnumerationParameter

Description

Lets you define an enumeration parameter. A enumeration parameter is defined by its name, data type and the list of all possible values.

The configured value can be accessed in the user code via the instance-specific runtime struct.

Example: days = {Monday,1},{Tuesday,2},{Wednesday,3},{Thursday,4},…

In a configuration dialog you choose between the days (Monday, …). In code, you use 'setDay(1)'.

Parent elements

Name	Purpose
Parameters on page 108	Lets you define type-specific configuration parameters: e.g., simple configuration parameters or enumeration parameters.

Child elements

Name	Multiplicity	Purpose
EnumerationValues on page 113	1..1	Definition of enumeration values

Attributes

Name	Use	Description
Name	required	Name of the element
Id	required	The ID has to be unique within the description file. It will be used for referencing.
DisplayName	optional	Name used for displaying only (e.g., in the property grid).
Description	optional	Documentation of the parameter. Will be displayed in property grid.
Category	optional	Optional, specific parent category of the parameter. Will be displayed in property grid.
AccessFlags	optional	Defines how and in which contexts the parameter can be accessed (default: editable, tunable, visible).
TraceType	optional	Defines the type of the tunable parameter within the experimentation tool (default: configuration parameter). Possible values: ▪ Configuration: The parameter is a configuration parameter. ▪ Measurement: The parameter is a measurement parameter.

SubstructureForVDF	optional	Defines an optional substructure for a tunable parameter within the variable description file.
		If not specified or empty, the parameter will be added at root level (root => parent element the parameter is generated at).
		Example: "subnode1:subsubnodeA" results in
		group "parent element"
		group "subnode1"
		group "subsubnodeA"
		tunable_Parameter
		{
		type: [...]
		alias: [...]
		flags: [...]
		range: [...]
		desc: [...]
		}
		endgroup
		endgroup
		endgroup

DataType	required	char[] parameters will not be included in the TRC file. Possible values: ▪ Bool ▪ UInt8 ▪ UInt16 ▪ UInt32 ▪ UInt64 ▪ Int8 ▪ Int16 ▪ Int32

Name	Use	Description
		▪ Int64 ▪ Float32 ▪ Float64 ▪ dsfloat ▪ Char[]
DefaultIndex	required	The value of the parameter will be initialized with the corresponding 'EnumerationValue'. 'DefaultIndex' is 1-based.

EnumerationValues

Description Lets you define the enumeration values for an enumeration parameter.

Parent elements

Name	Purpose
EnumerationParameter on page 111	Definition of an enumeration parameter

Child elements

Name	Multiplicity	Purpose
EnumerationValue on page 113	1..unbounded	Definition of an enumeration value

Attributes None

EnumerationValue

Description Lets you define an enumeration value for an enumeration parameter.

Parent elements

Name	Purpose
EnumerationValues on page 113	Definition of enumeration values

Child elements

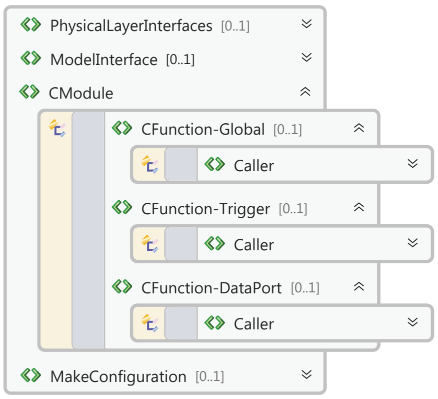
None

Attributes

Name	Use	Description
InternalValue	required	The internal value of the enumeration value. This must match the 'DataType' defined for the corresponding 'EnumerationParameter'. The value can also reference a user-defined constant.
DisplayName	required	Optional name used for displaying only (e.g., in the property grid).
TraceValue	optional	The value to be used in the context of addressing via an experimentation tool. When the enumeration parameter is tunable but no 'TraceValue' is defined, ConfigurationDesk tries to use the 'InternalValue' instead.

Hook Function Definitions

Hierarchy overview



Where to go from here

Information in this section

CModule

Description

Lets you define the user code module for the function block type.

Parent elements

Name	Purpose
CustomFunctionBlock on page 76	Definition of function block type

Child elements

Name	Multiplicity	Purpose
CFunction-DataPort on page 117	0..unbounded	Definition of a function call for a DataPort-related code hook.
CFunction-Global on page 115	0..unbounded	Definition of a function call for a global code hook
CFunction-Trigger on page 116	0..unbounded	Definition of a function call for a Function-related code hook.

Attributes

Name	Use	Description
Name	required	The name of the user code module without a file extension. A header (.h) and a source (.cpp) file with this name will be expected in the same directory as the XML file.

CFunction-Global

Description

Lets you define a function call for a global code hook. The name of the defined function has to be unique within the user code module.

Parent elements

Name	Purpose
CModule on page 115	Definition of the user code module

Child elements

Name	Multiplicity	Purpose
Caller (CFunction-Global) on page 117	1..1	Global code hook that is used.

Attributes

Name	Use	Description
Name	required	Name of the element

CFunction-Trigger

Description

Lets you define a function call for a Function-related code hook. The name of the defined function has to be unique within the user code module.

Parent elements

Name	Purpose
CModule on page 115	Definition of the user code module

Child elements

Name	Multiplicity	Purpose
Caller (CFunction-Function) on page 119	1..1	Function-related code hook that is used.

Attributes

Name	Use	Description
Name	required	Name of the element

CFunction-DataPort

Description

Lets you define a function call for a DataPort-related code hook. The name of the defined function has to be unique within the user code module.

Parent elements

Name	Purpose
CModule on page 115	Definition of the user code module

Child elements

Name	Multiplicity	Purpose
Caller (CFunction-DataPort) on page 119	1..1	DataPort-related code hook that is used.

Attributes

Name	Use	Description
Name	required	Name of the element

Caller (CFunction-Global)

Description

Lets you define the global code hook used for the corresponding function call.

Parent elements

Name	Purpose
CFunction-Global on page 115	Definition of a function call for a global code hook

Child elements

None

Attributes

Name	Use	Description
AccessPoint	required	Code-hook related AccessPoint Possible values: ▪ Creation: Contains code for instantiating driver objects. The AccessPoint will be called only once after starting the real-time application. ▪ PostCreation: Contains code for instantiating driver objects. The AccessPoint will be called only once after starting the real-time application and directly after code placed in access point "Creation". ▪ Init: Contains I/O initialization code that will be executed directly before the application is started. ▪ Stop: Contains code that will be executed when the SimEngine is stopped: e.g., the output of termination values. ▪ OneTimeInit: Contains code that will be executed after the initialization of RTOS-specific application parts: e.g., the setting of initial driver values. The AccessPoint will be called only once after starting the real-time application. ▪ Terminate: Contains code that will be executed when the SimEngine is terminated if there is a crash. In this AccessPoint, termination values can be set. ▪ Unload: Contains code that will be executed when the real-time application will be unloaded. Free memory resources on the heap that were reserved via malloc. ▪ Idle: Contains code that will be executed within the background task of the real-time application.
ApplicationProcess	(default: Own)	Defines in which application process the code-hook will be called. Default is 'Own'. Possible values: ▪ Own: Contains code for the application process the function block is (explicitly or implicitly) assigned to. ▪ Foreign: Contains code for foreign application processes the function block is not (explicitly or implicitly) assigned to but run in the same processing unit application as the function block. Only use this option if you are sure what you are doing.

Caller (CFunction-Function)

Description

Lets you define the Function-related code hook used for the corresponding function call.

Parent elements

Name	Purpose
CFunction-Trigger on page 116	Definition of a function call for a Function-related code hook.

Child elements

None

Attributes

Name	Use	Description
AccessPoint	required	Code-hook related AccessPoint Possible values: ▪ Entry: AccessPoint that will be called at the beginning of a function module. ▪ Exit: AccessPoint that will be called at the end of a function module.
FunctionId	required	The ID of a defined Function being related to this trigger.

Caller (CFunction-DataPort)

Description

Lets you define the DataPort-related code hook used for the corresponding function call.

Parent elements

Name	Purpose
CFunction-DataPort on page 117	Definition of a function call for a DataPort-related code hook.

Child elements

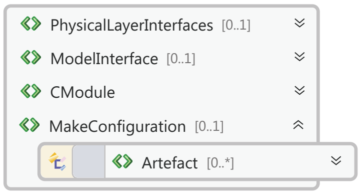
None

Attributes

Name	Use	Description
DataPortId	required	The ID of a defined DataPort being related to this function call.

Make Artifacts

Hierarchy overview



Where to go from here

Information in this section

MakeConfiguration

Description

Lets you define additional make artifacts.

For example: Driver-specific header files can be included in the generated IoCode_Data.h file. Driver-specific .so files can be added to the make process.

Parent elements

Name	Purpose
CustomFunctionBlock on page 76	Definition of function block type

Child elements

Name	Multiplicity	Purpose
Artefact on page 121	0..unbounded	Definition of an additional make artifact

Attributes None

Artefact

Description Lets you define an additional make artifact.

Parent elements

Name	Purpose
MakeConfiguration on page 120	Make configuration

Child elements None

Attributes

Name	Use	Description
Name	required	The name of the artifact file. In the case of a dSPACE RTLib library, use the basename only (libbasename.so → basename). For other libraries, qualify the full library file name and also a corresponding search path via the 'Directory' attribute.
Directory	optional	The directory the artifact is located in. This attribute is mandatory if the artifact is not part of the dSPACE installation, e.g. if a CustomDynamicLibrary is specified. The value can be either a relative or an absolute path.
Type	required	The artifact type. Header files will be included in the IoCode_Data.h file. Library or object files will be included in the make command. Possible values: ▪ Headerfile: A header file will be included in the IoCode_Data.h file. ▪ Library: A library will be included in the make command. ▪ CustomDynamicLibrary:

Name	Use	Description
		A custom dynamic library will be dynamically linked in the make command and included in the RTA file. ▪ Objectfile: An object file will be included in the make command. ▪ Sourcefile ▪ Custom: A custom file will be included in the RTA file.

ConfigurationDesk Glossary

Introduction

The glossary briefly explains the most important expressions and naming conventions used in the ConfigurationDesk documentation.

Where to go from here

Information in this section

A

Application There are two types of applications in ConfigurationDesk:

- A part of a ConfigurationDesk project: ConfigurationDesk application.
- An application that can be executed on dSPACE real-time hardware: real-time application.

Application process A component of a processing unit application. An application process contains one or more tasks.

Application process component A component of an application process. The following application process components are available in the **Components** subfolder of an application process:

- Behavior models that are assigned to the application process, including their predefined tasks, runnable functions, and events.
- Function blocks that are assigned to the application process.

AutomationDesk A dSPACE software product for creating and managing any kind of automation tasks. Within the dSPACE tool chain, it is mainly used for automating tests on dSPACE hardware.

AUTOSAR system description file An AUTOSAR XML (ARXML) file that describes a system according to AUTOSAR. A system is a combination of a hardware topology, a software architecture, a network communication, and information on the mappings between these elements. The described network communication usually consists of more than one bus system (e.g., CAN, LIN, FlexRay).

B

Basic PDU A general term used in the documentation to address all the PDUs the Bus Manager supports, except for container IPDUs, multiplexed IPDUs, and secured IPDUs. Basic PDUs are represented by the ▶ or ◧ symbol in tables and browsers. The Bus Manager provides the same functionalities for all basic PDUs, such as ISignal IPDUs or NMPDUs.

Behavior model A model that contains the control algorithm for a controller (function prototyping system) or the algorithm of the controlled system (hardware-in-the-loop system). It does not contain I/O functionality nor access to the hardware. Behavior models can be modeled, for example, in MATLAB/Simulink by using Simulink Blocksets and Toolboxes from the MathWorks®.

You can add Simulink behavior models to a ConfigurationDesk application. You can also add code container files containing a behavior model such as Functional Mock-up Units, or Simulink implementation containers to a ConfigurationDesk application.

Bidirectional signal port A signal port that is independent of a data direction or current flow. This port is used, for example, to implement bus communication.

BSC file A bus simulation container file that is generated with the Bus Manager and contains the configured bus communication of one application process.

Build Configuration table A pane that lets you create build configuration sets and configure build settings, for example, build options, or the build and download behavior.

Build Log Viewer A pane that displays messages and warnings during the build process.

Build process A process that generates an executable real-time application based on your ConfigurationDesk application that can be run on a SCALEXIO system or MicroAutoBox III system. The build process can be controlled and configured via the Build Log Viewer. If the build process is successfully finished, the build result files (build results) are added to the ConfigurationDesk application.

Build results The files that are created during the build process. Build results are named after the ConfigurationDesk application and the application process from which they originate. You can access the build results in the Project Manager.

Bus access The representation of a run-time communication cluster. By assigning one or more bus access requests to a bus access, you specify which communication clusters form one run-time communication cluster.

In ConfigurationDesk, you can use a bus function block (CAN, LIN) to implement a bus access. The hardware resource assignment of the bus function block specifies the bus channel that is used for the bus communication.

Bus access request The representation of a request regarding the bus access. There are two sources for bus access requests:

- At least one element of a communication cluster is assigned to the Simulated ECUs, Inspection, or Manipulation part of a bus configuration. The related bus access requests contain the requirements for the bus channels that are to be used for the cluster's bus communication.

- A frame gateway is added to the **Gateways** part of a bus configuration. Each frame gateway provides two bus access requests that are required to specify the bus channels for exchanging bus communication.

Bus access requests are automatically included in BSC files. To build a real-time application, each bus access request must be assigned to a bus access.

Bus Access Requests table A pane that lets you access bus access requests of a ConfigurationDesk application and assign them to bus accesses.

Bus configuration A Bus Manager element that implements bus communication in a ConfigurationDesk application and lets you configure it for simulation, inspection, and/or manipulation purposes. The required bus communication elements must be specified in a communication matrix and assigned to the bus configuration. Additionally, a bus configuration lets you specify gateways for exchanging bus communication between communication clusters. A bus configuration can be accessed via specific tables and its related Bus Configuration function block.

Bus Configuration Function Ports table A pane that lets you access and configure function ports of bus configurations.

Bus Configurations table A pane that lets you access and configure bus configurations of a ConfigurationDesk application.

Bus Inspection Features table A pane that lets you access and configure bus configuration features of a ConfigurationDesk application for inspection purposes.

Bus Manager

- Bus Manager in ConfigurationDesk

  A ConfigurationDesk component that lets you configure bus communication and implement it in real-time applications or generate bus simulation containers.
- Bus Manager (stand-alone)

  A dSPACE software product based on ConfigurationDesk that lets you configure bus communication and generate bus simulation containers.

Bus Manipulation Features table A pane that lets you access and configure bus configuration features of a ConfigurationDesk application for manipulation purposes.

Bus simulation container A container that contains bus communication configured with the Bus Manager. Bus simulation container (BSC) files can be used in the VEOS Player and in ConfigurationDesk. In the VEOS Player, they let you implement the bus communication in an offline simulation application.

In ConfigurationDesk, they let you implement the bus communication in a real-time application independently from the Bus Manager.

Bus Simulation Features table A pane that lets you access and configure bus configuration features of a ConfigurationDesk application for simulation purposes.

Buses Browser A pane that lets you display and manage the communication matrices of a ConfigurationDesk application. For example, you can access communication matrix elements and assign them to bus configurations. This pane is available only if you work with the Bus Manager.

C

Cable harness A bundle of cables that provides the connection between the I/O connectors of the real-time hardware and the external devices, such as the ECUs to be tested. In ConfigurationDesk, it is represented by an external cable harness component.

CAFX file A ConfigurationDesk application fragment file that contains signal chain elements that were exported from a user-defined working view or the Temporary working view of a ConfigurationDesk application. This includes the elements' configuration and the mapping lines between them.

CDL file A ConfigurationDesk application file that contains links to all the documents related to an application.

Channel multiplication A feature that allows you to enhance the max. current or max. voltage of a single hardware channel by combining several channels. ConfigurationDesk uses a user-defined value to calculate the number of hardware channels needed. Depending on the function block type, channel multiplication is provided either for current enhancement (two or more channels are connected in parallel) or for voltage enhancement (two or more channels are connected in series).

Channel request A channel assignment required by a function block. ConfigurationDesk determines the type(s) and number of channels required for a function block according to the assigned channel set, the function block features, the block configuration and the required physical ranges. ConfigurationDesk provides a set of suitable and available hardware resources for each channel request. This set is produced according to the hardware topology added to the active ConfigurationDesk application. You have to assign each channel request to a specific channel of the hardware topology.

Channel set A number of channels of the same channel type located on the same I/O board or I/O unit. Channels in a channel set can be combined, for example, to provide a signal with channel multiplication.

Channel type A term to indicate all the hardware resources (channels) in the hardware system that provide exactly the same characteristics. Examples for channel type names: Flexible In 1, Digital Out 3, Analog In 1. An I/O board in a hardware system can have channel sets of several channel types. Channel sets of one channel type can be available on different I/O boards.

Cluster Communication cluster.

Common Program Data folder A standard folder for application-specific configuration data that is used by all users.

`%PROGRAMDATA%\dSPACE\<InstallationGUID>\<ProductName>`

or

`%PROGRAMDATA%\dSPACE\<ProductName>\<VersionNumber>`

Communication cluster A communication network of network nodes that are connected to the same physical channels and share the same bus protocol and address range.

Communication matrix A file that defines the communication of a bus network. It can describe the bus communication of one communication cluster or a bus network consisting of different bus systems and clusters. Files of various file formats can be used as a communication matrix: For example, AUTOSAR system description files, DBC files, LDF files, and FIBEX files.

Communication package A package that bundles Data Inport blocks which are connected to Data Outport blocks. Hence, it also bundles the signals that are received by these blocks. If Data Inport blocks are executed within the same task and belong to the same communication package, their data inports are read simultaneously. If Data Outport blocks that are connected to the Data Inport blocks are executed in the same task, their output signals are sent simultaneously in one data package. Thus, communication packages guarantee simultaneous signal updates within a running task (data snapshot).

Configuration port A port that lets you create the signal chain for the bus communication implemented in a Simulink behavior model. The following configuration ports are available:

- The configuration port of a Configuration Port block.
- The Configuration port of a CAN, LIN, or FlexRay function block.

To create the signal chain for bus communication, the configuration port of a Configuration Port block must be mapped to the Configuration port of a CAN, LIN, or FlexRay function block.

Configuration Port block A model port block that is created in ConfigurationDesk during model analysis for each of the following blocks found in the Simulink behavior model:

- RTICANMM ControllerSetup block
- RTILINMM ControllerSetup block
- FLEXRAYCONFIG UPDATE block

Configuration Port blocks are also created for bus simulation containers. A Configuration Port block provides a configuration port that must be mapped to the Configuration port of a **CAN**, **LIN**, or **FlexRay** function block to create the signal chain for bus communication.

ConfigurationDesk application A part of a ConfigurationDesk project that represents a specific implementation. You can work with only one application at a time, and that application must be activated.

An application can contain:

- Device topology
- Hardware topology
- Model topology
- Communication matrices
- External cable harness
- Build results (after a successful build process has finished)

You can also add folders with application-specific files to an application.

ConfigurationDesk model interface The part of the model interface that is available in ConfigurationDesk. This specific term is used to explicitly distinguish between the model interface in ConfigurationDesk and the model interface in Simulink.

Conflict A result of conflicting configuration settings that is displayed in the Conflicts Viewer. ConfigurationDesk allows flexible configuration without strict constraints. This lets you work more freely, but it can lead to conflicting configuration settings. ConfigurationDesk automatically detects conflicts and provides the **Conflicts Viewer** to display and help resolve them. Before you build a real-time application, you have to resolve at least the most severe conflicts (e.g., errors that abort the build process) to get proper build results.

Conflicts Viewer A pane that displays the configuration conflicts that exist in the active ConfigurationDesk application. You can resolve most of the conflicts directly in the **Conflicts Viewer**.

Container IPDU A term according to AUTOSAR. An IPDU that contains one or more other IPDUs (i.e., contained IPDUs). When a container IPDU is mapped to a frame, all its contained IPDUs are included in that frame as well.

ControlDesk A dSPACE software product for managing, instrumenting and executing experiments for ECU development. ControlDesk also supports calibration, measurement and diagnostics access to ECUs via standardized protocols such as CCP, XCP, and ODX.

CTLGZ file A ZIP file that contains a V-ECU implementation. CTLGZ files are exported by TargetLink or SystemDesk. You can add a V-ECU implementation based on a CTLGZ file to the model topology just like adding a Simulink model based on an SLX file.

Cycle time restriction A value of a runnable function that indicates the sample time the runnable function requires to achieve correct results. The cycle time restriction is indicated by the **Period** property of the runnable function in the Properties Browser.

D

Data inport A port that supplies data from ConfigurationDesk's function outports to the behavior model.

In a multimodel application, data inports also can be used to provide data from a data outport associated to another behavior model (model communication).

Data outport A port that supplies data from behavior model signals to ConfigurationDesk's function inports.

In a multimodel application, data outports also can be used to supply data to a data inport associated to another behavior model (model communication).

DBC file A Data Base Container file that describes CAN or LIN bus systems. Because the DBC file format was primarily developed to describe CAN networks, it does not support definitions of LIN masters and schedules.

Device block A graphical representation of devices from the device topology in the signal chain. It can be mapped to function blocks via device ports.

Device connector A structural element that lets you group device pins in a hierarchy in the External Device Connectors table to represent the structure of the real connector of your external device.

Device pin A representation of a connector pin of your external device. Device ports are assigned to device pins. ConfigurationDesk can use the device pin assignment together with the hardware resource assignment and the device port mapping to calculate the external cable harness.

Device port An element of a device topology that represents the signal of an external device in ConfigurationDesk.

Device port group A structural element of a device topology that can contain device ports and other device port groups.

Device topology A component of a ConfigurationDesk application that represents external devices in ConfigurationDesk. You can create a device topology from scratch or easily extend an existing device topology. You can also merge device topologies to extend one. To edit or create device topologies independently of ConfigurationDesk, you can export and import DTFX and XLSX files.

Documents folder A standard folder for user-specific documents.

`%USERPROFILE%\Documents\dSPACE\<ProductName>\`
`<VersionNumber>`

DSA file A dSPACE archive file that contains a ConfigurationDesk application and all the files belonging to it as one unit. It can later be imported to another ConfigurationDesk project.

dSPACE Help The dSPACE online help that contains all the relevant user documentation for dSPACE products. Via the **F1** key or the **Help** button in the dSPACE software you get context-sensitive help on the currently active context.

dSPACE Log A collection of errors, warnings, information, questions, and advice issued by all dSPACE products and connected systems over more than one session.

DTFX file A device topology export file that contains information on the interface to the external devices, such as the ECUs to be tested. The information includes details of the available device ports, their characteristics, and the assigned pins.

E

ECHX file An external cable harness file that contains the wiring information for the external cable harness. The external cable harness is the connection between the I/O connectors of the real-time hardware and the devices to be tested, for example, ECUs.

ECU Abbreviation of *electronic control unit*.

An embedded computer system that consists of at least one CPU and associated peripherals. An ECU contains communication controllers and communication connectors, and usually communicates with other ECUs of a bus network. An ECU can be member of multiple bus systems and communication clusters.

ECU application An application that is executed on an ECU. In ECU interfacing scenarios, parts of the ECU application can be accessed (e.g., by a real-time application) for development and testing purposes.

ECU function A function of an ECU application that is executed on the ECU. In ECU interfacing scenarios, an ECU function can be accessed by functions that are part of a real-time application, for example.

ECU Interface Manager A dSPACE software product for preparing ECU applications for ECU interfacing. The ECU Interface Manager can generate ECU interface container (EIC) files to be used in ConfigurationDesk.

ECU interfacing A generic term for methods and tools to read and/or write individual ECU functions and variables of an ECU application. In ECU interfacing scenarios, you can access ECU functions and variables for development and testing purposes while the ECU application is executed on the ECU. For example, you can perform ECU interfacing with SCALEXIO systems or MicroAutoBox III systems to access individual ECU functions by a real-time application.

EIC file An ECU interface container file that is generated with the ECU Interface Manager and describes an ECU application that is configured for ECU interfacing. You can import EIC files to ConfigurationDesk to perform ECU interfacing with SCALEXIO systems or MicroAutoBox III systems.

Electrical interface unit A segment of a function block that provides the interface to the external devices and to the real-time hardware (via hardware resource assignment). Each electrical interface unit of a function block usually needs a channel set to be assigned to it.

Event A component of a ConfigurationDesk application that triggers the execution of a task. The following event types are available:

- Timer event
- I/O event
- Software event

Event port An element of a function block. The event port can be mapped to a runnable function port for modeling an asynchronous task.

Executable application The generic term for real-time applications and offline simulation applications. In ConfigurationDesk, an executable application is always a real-time application since ConfigurationDesk does not support offline simulation applications.

Executable application component A component of an executable application. The following components can be part of an executable application:

- Imported behavior models including predefined tasks, runnable functions, and events. You can assign these behavior models to application processes via drag & drop or by selecting the **Assign Model** command from the context menu of the relevant application process.
- Function blocks added to your ConfigurationDesk application including associated I/O events. Function blocks are assigned to application processes via their model port mapping.

Executable Application table A pane that lets you model executable applications (i.e., real-time applications) and the tasks used in them.

EXPSWCFG file An experiment software configuration file that contains configuration data for automotive fieldbus communication. It is created during the build process and contains the data in XML format.

External cable harness A component of a ConfigurationDesk application that contains the wiring information for the external cable harness (also known as cable harness). It contains only the logical connections and no additional information such as cable length, cable diameters, dimensions or the arrangement of connection points, etc. It can be calculated by ConfigurationDesk or imported from a file so that you can use an existing cable harness and do not have to build a new one. The wiring information can be exported to an ECHX file or XLSX file.

External device A device that is connected to the dSPACE hardware, such as an ECU or external load. The external device topology is the basis for using external devices in the signal chain of a ConfigurationDesk application.

External Device Browser A pane that lets you display and manage the device topology of your active ConfigurationDesk application.

External Device Configuration table A pane that lets you access and configure the most important properties of device topology elements via table.

External Device Connectors table A pane that lets you specify the representation of the physical connectors of your external device including the device pin assignment.

F

FIBEX file An XML file according the ASAM MCD-2 NET standard (also known as Field Bus Exchange Format) defined by ASAM. The file can describe more than one bus system (e.g., CAN, LIN, FlexRay). It is used for data exchange between different tools that work with message-oriented bus communication.

Find Results Viewer A pane that displays the results of searches you performed via the Find command.

FMU file A Functional Mock-up Unit file that describes and implements the functionality of a model. It is an archive file with the file name extension FMU. The FMU file contains:

- The functionality defined as a set of C functions provided either in source or in binary form.
- The model description file (`modelDescription.xml`) with the description of the interface data.
- Additional resources needed for simulation.

You can add an FMU file to the model topology just like adding a Simulink model based on an SLX file.

Frame A piece of information of a bus communication. It contains an arbitrary number of non-overlapping PDUs and the data length code (DLC). CAN frames and LIN frames can contain only one PDU. To exchange a frame via bus channels, a frame triggering is needed.

Frame triggering An instance of a frame that is exchanged via a bus channel. It includes transmission information of the frame (e.g., timings, ID, sender, receiver). The requirements regarding the frame triggerings depend on the bus system (CAN, LIN, FlexRay).

Function block A graphical representation in the signal chain that is instantiated from a function block type. A function block provides the I/O functionality and the connection to the real-time hardware. It serves as a container for functions, for electrical interface units and their logical signals. The function block's ports (function ports and/or signal ports), provide the interfaces to the neighboring blocks in the signal chain.

Function block type A software plug-in that provides a specific I/O functionality. Every function block type has unique features which are different from other function block types.

To use a function block type in your ConfigurationDesk application, you have to create an instance of it. This instance is called a function block. Instances of function block types can be used multiple times in a ConfigurationDesk application. The types and their instantiated function blocks are displayed in the function library of the Function Browser.

Function Browser A pane that displays the function library in a hierarchical tree structure. Function block types are grouped in function classes. Instantiated function blocks are added below the corresponding function block type.

Function inport A function port that inputs the values from the behavior model to the function block to be processed by the function.

Function library A collection of function block types that allows access to the I/O functionality in ConfigurationDesk. The I/O functionality is based on function block types. The function library provides a structured tree view on the available function block types. It is displayed in the Function Browser.

Function outport A function port that outputs the value of a function to be used in the behavior model.

Function port An element of a function block that provides the interface to the behavior model via model port blocks.

Functional Mock-up Unit An archive file that describes and implements the functionality of a model based on the Functional Mock-up Interface (FMI) standard.

G

Global working view The default working view that always contains all signal chain elements.

H

Hardware resource A hardware element (normally a channel on an I/O board or I/O unit) which is required to execute a function block. A hardware resource can be localized unambiguously in a hardware system. Every hardware resource has specific characteristics. A function block therefore needs a hardware resource that matches the requirements of its functionality. This means that not every function block can be executed on every hardware resource. There could be limitations on a function block's features and/or the physical ranges.

Hardware resource assignment An action that assigns the electrical interface unit of a function block to one or more hardware resources. Function blocks can be assigned to any hardware resource which is suitable for the functionality and available in the hardware topology of your ConfigurationDesk application.

Hardware Resource Browser A pane that lets you display and manage all the hardware components of the hardware topology that is contained in your active ConfigurationDesk application in a hierarchical structure.

Hardware topology A component of a ConfigurationDesk application that contains information on a specific hardware system which can be used with ConfigurationDesk. It provides information on the components of the system, such as channel type and slot numbers. It can be scanned automatically from a registered platform, created in ConfigurationDesk's Hardware Resource Browser from scratch, or imported from an HTFX file.

HTFX file A file containing the hardware topology after an explicit export. It provides information on the components of the system and also on the channel properties, such as board and channel types and slot numbers.

I/O event An asynchronous event triggered by I/O functions. You can use I/O events to trigger tasks in your application process asynchronously. You can assign the events to the tasks via drag & drop, via the Properties Browser if you have selected a task, or via the Assign Event command from the context menu of the relevant task.

Interface model A temporary Simulink model that contains blocks from the Model Interface Blockset. ConfigurationDesk initiates the creation of an interface model in Simulink. You can copy the blocks with their identities from the interface model and paste them into an existing Simulink behavior model.

Interpreter A pane that lets you run Python scripts and execute line-based commands.

Inverse model port block A model port block that has the same configuration (same name, same port groups, and port names) but the inverse data direction as the original model port block from which it was created.

IOCNET Abbreviation of I/O carrier network.

A dSPACE proprietary protocol for internal communication in a SCALEXIO system between the real-time processors and I/O units. The IOCNET lets you connect more than 100 I/O nodes and place the parts of your SCALEXIO system long distances apart.

IPDU Abbreviation of interaction layer protocol data unit.

A term according to AUTOSAR. An IPDU contains the communication data that is routed from the interaction layer to a lower communication layer and vice versa. An IPDU can be implemented, for example, as an ISignal IPDU, multiplexed IPDU, or container IPDU.

ISignal A term according to AUTOSAR. A signal of the interaction layer that contains communication data as a coded signal value. To transmit the communication data on a bus, ISignals are instantiated and included in ISignal IPDUs.

ISignal IPDU A term according to AUTOSAR. An IPDU whose communication data is arranged in ISignals. ISignal IPDUs allow the exchange of ISignals between different network nodes.

L

LDF file A LIN description file that describes networks of the LIN bus system according to the LIN standard.

LIN master A member of a LIN communication cluster that is responsible for the timing of LIN bus communication. A LIN master provides one LIN master task and one LIN slave task. The LIN master task transmits frame headers on the bus, and provides LIN schedule tables and LIN collision resolver tables. The LIN slave task transmits frame responses on the bus. A LIN cluster must contain exactly one LIN master.

LIN schedule table A table defined for a LIN master that contains the transmission sequence of frame headers on a LIN bus. For each LIN master, several LIN schedule tables can be defined.

LIN slave A member of a LIN communication cluster that provides only a LIN slave task. The LIN slave task transmits frame responses on the bus when they are triggered by a frame header. The frame header is sent by a LIN master. A LIN cluster can contain several LIN slaves.

Local Program Data folder A standard folder for application-specific configuration data that is used by the current, non-roaming user.

```
%USERPROFILE%\AppData\Local\dSPACE\<InstallationGUID>\
<ProductName>
```

Logical signal An element of a function block that combines all the signal ports which belong together to provide the functionality of the signal. Each logical signal causes one or more channel requests. Channel requests are available after you have assigned a channel set to the logical signal.

Logical signal chain A term that describes the logical path of a signal between an external device and the behavior model. The main elements of the logical signal chain are represented by different graphical blocks (device blocks, function blocks and model port blocks). Every block has ports to provide the mapping to neighboring blocks.

In the documentation, usually the short form 'signal chain' is used instead.

M

MAP file A file that maps symbolic names to physical addresses.

Mapping line A graphical representation of a connection between two ports in the signal chain. You can draw mapping lines in a working view.

MCD file A model communication description file that is used to implement a multimodel application. It lets you add several behavior models that were separated from an overall model to the model topology.

The MCD file contains information on the separated models and information on the connections between them. The file is generated with the Model Separation Setup Block in MATLAB/Simulink. The block resides in the Model Interface Blockset (dsmpblib) from dSPACE.

MDL file A Simulink model file that contains the behavior model. You can add an MDL file to your ConfigurationDesk application.

As of MATLAB® R2012a, the file name extension for the Simulink model file has been changed from MDL to SLX by The MathWorks®.

Message Viewer A pane that displays a history of all error and warning messages that occur during work with ConfigurationDesk.

Model analysis A process that analyzes the model to determine the interface of a behavior model. You can select one of the following commands:

- Analyze Simulink Model (Model Interface Only)

  Analyzes the interface of a behavior model. The model topology of your active ConfigurationDesk application is updated with the properties of the analyzed behavior model.
- Analyze Simulink Model (Including Task Information)

  Analyzes the model interface and the elements of the behavior model that are relevant for the task configuration. The task configuration in ConfigurationDesk is then updated accordingly.

Model Browser A pane that lets you display and access the model topology of an active ConfigurationDesk application. The Model Browser provides access to all the model port blocks available in the behavior models which are linked to a ConfigurationDesk application. The model elements are displayed in a hierarchy, starting with the model roots. Below the model root, all the subsystems containing model port blocks are displayed as well as the associated model port blocks.

Model communication The exchange of signal data between the models within a multimodel application. To set up model communication, you must use a mapping line to connect a data outport (sending model) to a data inport (receiving model). The best way to set up model communication is using the Model Communication Browser.

Model Communication Browser A pane that lets you open and browse working views like the **Signal Chain Browser**, but shows only the Data Outport and Data Inport blocks and the mapping lines between them.

Model Communication Package table A pane that lets you create and configure model communication packages which are used for model communication in multimodel applications.

Model implementation An implementation of a behavior model. It can consist of source code files, precompiled objects or libraries, variable description files and a description of the model's interface. Specific model implementation types are, for example, model implementation containers, such as Functional Mock-up Units or Simulink implementation containers.

Model implementation container A file archive that contains a model implementation. Examples are FMUs, SIC files, and VECU files.

Model interface An interface that connects ConfigurationDesk with a behavior model. This interface is part of the signal chain and is implemented via model port blocks. The model port blocks in ConfigurationDesk can provide the interface to:

- Model port blocks (from the Model Interface Package for Simulink) in a Simulink behavior model. In this case, the model interface is also called ConfigurationDesk model interface to distinguish it from the Simulink model interface available in the Simulink behavior model.
- Different types of model implementations based on code container files, e.g., Simulink implementation containers, Functional Mock-up Units, and V-ECU implementations.

Model Interface Package for Simulink A dSPACE software product that lets you specify the interface of a behavior model that you can directly use in ConfigurationDesk. You can also create a code container file (SIC file) that contains the model code of a Simulink behavior model. The SIC file can be used in ConfigurationDesk and VEOS Player.

Model port An element of a model port block. Model ports provide the interface to the function ports and to other model ports (in multimodel applications).

These are the types of model ports:

- Data inport
- Data outport
- Runnable function port
- Configuration port

Model port block A graphical representation of the ConfigurationDesk model interface in the signal chain. Model port blocks contain model ports that can be mapped to function blocks to provide the data flow between the function blocks in ConfigurationDesk and the behavior model. The model ports can also be mapped to the model ports of other model port blocks with data inports or data outports to set up model communication. Model port blocks are available in different types and can provide different port types:

- Data port blocks with data inports and data outports
- Runnable Function blocks with runnable function ports
- Configuration Port blocks with configuration ports. Configuration Port blocks are created during model analysis for each of the following blocks found in the Simulink behavior model:
  - RTICANMM ControllerSetup block
  - RTILINMM ControllerSetup block
  - FLEXRAYCONFIG UPDATE block

  Configuration Port blocks are also created for bus simulation containers.

Model Separation Setup Block A block that is contained in the Model Interface Package for Simulink. It is used to separate individual models from an overall model in MATLAB/Simulink. Additionally, model separation generates a model communication description file (MCD file) that contains information on the separated models and their connections. You can use this MCD file in ConfigurationDesk.

Model topology A component of a ConfigurationDesk application that contains information on the subsystems and the associated model port blocks of all the behavior models that have been added to a ConfigurationDesk application.

Model-Function Mapping Browser A pane that lets you create and update signal chains for Simulink behavior models. It directly connects them to I/O functionality in ConfigurationDesk.

MTFX file A file containing a model topology when explicitly exported. The file contains information on the interface to the behavior model, such as the implemented model port blocks including their subsystems and where they are used in the model.

Multicore real-time application A real-time application that is executed on several cores of one PU of the real-time hardware.

Multimodel application A real-time application that executes several behavior models in parallel on dSPACE real-time hardware (SCALEXIO or MicroAutoBox III).

Multiplexed IPDU A term according to AUTOSAR. An IPDU that consists of one dynamic part, a selector field, and one optional static part. Multiplexing is used to transport varying ISignal IPDUs via the same bytes of a multiplexed IPDU.

- The dynamic part is one ISignal IPDU that is selected for transmission at run time. Several ISignal IPDUs can be specified as dynamic part alternatives. One of these alternatives is selected for transmission.

- The selector field value indicates which ISignal IPDU is transmitted in the dynamic part during run time. For each selector field value, there is one corresponding ISignal IPDU of the dynamic part alternatives. The selector field value is evaluated by the receiver of the multiplexed IPDU.
- The static part is one ISignal IPDU that is always transmitted.

Multi-PU application Abbreviation of multi-processing-unit application. A multi-PU application is a real-time application that is partitioned into several processing unit applications. Each processing unit application is executed on a separate PU of the real-time hardware. The processing units are connected via IOCNET and can be accessed from the same host PC.

N

Navigation bar An element of ConfigurationDesk's user interface that lets you switch between view sets.

Network node A term that describes the bus communication of an ECU for only one communication cluster.

O

Offline simulation A purely PC-based simulation scenario without a connection to a physical system, i.e., neither simulator hardware nor ECU hardware prototypes are needed. Offline simulations are independent from real time and can run on VEOS.

Offline simulation application An application that runs on VEOS to perform offline simulation. An offline simulation application can be built with the VEOS Player and the resulting OSA file can be downloaded to VEOS.

OSA file An offline simulation application file that is built with the VEOS Player and can be downloaded to VEOS to perform offline simulation.

P

Parent port A port that you can use to map multiple function ports and model ports. All child ports with the same name are mapped. ConfigurationDesk enforces the mapping rules and allows only mapping lines which agree with them.

PDU Abbreviation of protocol data unit.

A term according to AUTOSAR. A PDU transports data (e.g., control information or communication data) via one or multiple network layers according to the AUTOSAR layered architecture. Depending on the involved layers and the function of a PDU, various PDU types can be distinguished, e.g., ISignal IPDUs, multiplexed IPDUs, and NMPDUs.

Physical signal chain A term that describes the electrical wiring of external devices (ECU and loads) to the I/O boards of the real-time hardware. The physical signal chain includes the external cable harness, the pinouts of the connectors and the internal cable harness.

Pins and External Wiring table A pane that lets you access the external wiring information

Platform A dSPACE real-time hardware system that can be registered and displayed in the Platform Manager.

Platform Manager A pane that lets you handle registered hardware platforms. You can download, start, and stop real-time applications via the Platform Manager. You can also update the firmware of your SCALEXIO system or MicroAutoBox III system.

Preconfigured application process An application process that was created via the Create preconfigured application process command. If you use the command, ConfigurationDesk creates new tasks for each runnable function provided by the model which is not assigned to a predefined task. ConfigurationDesk assigns the corresponding runnable function and (for periodic tasks) timer events to the new tasks. The tasks are preconfigured (e.g., DAQ raster name, period).

Processing Resource Assignment table A pane that lets you configure and inspect the processing resources in an executable application. This table is useful especially for multi-processing-unit applications.

Processing unit application A component of an executable application. A processing unit application contains one or more application processes.

Project A container for ConfigurationDesk applications and all project-specific documents. You must define a project or open an existing one to work with ConfigurationDesk. Projects are stored in a project root folder.

Project Manager A pane that provides access to ConfigurationDesk projects and applications and all the files they contain.

Project root folder A folder on your file system to which ConfigurationDesk saves all project-relevant data, such as the applications and documents of a project. Several projects can use the same project root folder. ConfigurationDesk uses the Documents folder as the default project root folder. You can specify further project root folders. Each can be made the default project root folder.

Properties Browser A pane that lets you access the properties of selected elements.

PU Abbreviation of processing unit.

A hardware assembly that consists of a motherboard or a dSPACE processor board, possibly additional interface hardware for connecting I/O boards, and an enclosure, i.e., a SCALEXIO Real-Time PC.

R

Real-time application An application that can be executed in real time on dSPACE real-time hardware. The real-time application is the result of a build process. It can be downloaded to real-time hardware via an RTA file. There are different types of real-time applications:

- Single-core real-time application.
- Multicore real-time application.
- Multi-PU application.

Restbus simulation A simulation method to test real ECUs by connecting them to a simulator that simulates the other ECUs in the communication clusters.

RTA file A real-time application file. An RTA file is an executable object file for processor boards. It is created during the build process. After the build process it can be downloaded to the real-time hardware.

Runnable function A function that is called by a task to compute results. A model implementation provides a runnable function for its base rate task. This runnable function can be executed in a task that is triggered by a timer event. In addition, a Simulink behavior model provides a runnable function for each Hardware-Triggered Runnable Function block contained in the Simulink behavior model. This runnable function is suitable for being executed in an asynchronous task.

Runnable Function block A type of model port block. A Runnable Function block provides a runnable function port that can be mapped to an event port of a function block for modeling an asynchronous task.

Runnable function port An element of a Runnable Function block. The runnable function port can be mapped to an event port of a function block for modeling an asynchronous task.

RX Communication data that is received by a bus member.

S

SCALEXIO system A dSPACE hardware-in-the-loop (HIL) system consisting of one or more real-time industry PCs (PUs), I/O boards, and I/O units. They communicate with each other via the IOCNET. The system simulates the environment to test an ECU. It provides the sensor signals for the ECU, measures the signals of the ECU, and provides the power (battery voltage) for the ECU and a bus interface for restbus simulation.

SDF file A system description file that contains information on the CPU name(s), the corresponding object file(s) to be downloaded and the corresponding variable description file(s). It is created during the build process.

Secured IPDU A term according to AUTOSAR. An IPDU that secures the payload of another PDU (i.e., authentic IPDU) for secure onboard communication (SecOC). The secured IPDU contains the authentication information that is used to secure the authentic IPDU's payload. If the secured IPDU is configured as a cryptographic IPDU, the secured IPDU and the authentic IPDU are mapped to different frames. If the secured IPDU is not configured as a cryptographic IPDU, the authentic IPDU is directly included in the secured IPDU.

SIC file A Simulink implementation container file that contains the model code of a Simulink behavior model. The SIC file can be used in ConfigurationDesk and in VEOS Player.

Signal chain A term used in the documentation as a short form for logical signal chain. Do not confuse it with the physical signal chain.

Signal Chain Browser A pane that lets you open and browse working views such as the Global working view or user-defined working views.

Signal inport A signal port that represents an electrical connection point of a function block which provides signal measurement (= input) functionality.

Signal outport A signal port that represents an electrical connection point of a function block which provides signal generation (= output) functionality.

Signal port An element of a function block that provides the interface to external devices (e.g., ECUs) via device blocks. It represents an electrical connection point of a function block.

Signal reference port A signal port that represents a connection point for the reference potential of inports, outports and bidirectional ports. For example: With differential signals, this is a reference signal, with single-ended signals, it is the ground signal (GND).

Simulink implementation container A container that contains the model code of a Simulink behavior model. A Simulink implementation container is generated from a Simulink behavior model by using the Model Interface Package for Simulink. The file name extension of a Simulink implementation container is SIC.

Simulink model interface The part of the model interface that is available in the connected Simulink behavior model.

Single-core real-time application An executable application that is executed on only one core of the real-time hardware.

Single-PU system Abbreviation of single-processing-unit system.

A system consisting of exactly one PU and the directly connected I/O units and I/O routers.

SLX file A Simulink model file that contains the behavior model. You can add an SLX file to your ConfigurationDesk application.

As of MATLAB® R2012a, the file name extension for the Simulink model file has been changed from MDL to SLX by The MathWorks®.

Software event An event that is activated from within a task to trigger another subordinate task. Consider the following example: A multi-tasking Simulink behavior model has a base rate task with a sample time = 1 ms and a periodic task with a sample time = 4 ms. In this case, the periodic task is triggered on every fourth execution of the base rate task via a software event. Software events are available in ConfigurationDesk after model analysis.

Source Code Editor A Python editor that lets you open and edit Python scripts that you open from or create in a ConfigurationDesk project in a window in the working area. You cannot run a Python script in a Source Code Editor window. To run a Python script you can use the Run Script command in the Interpreter or on the Automation ribbon or the Run context menu command in the Project Manager.

Structured data port A hierarchically structured port of a Data Inport block or a Data Outport block. Each structured data port consists of more structured and/or unstructured data ports. The structured data ports can consist of signals with different data types (single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, Boolean).

SystemDesk A dSPACE software product for development of distributed automotive electrics/electronics systems according to the AUTOSAR approach.

SystemDesk is able to provide a V-ECU implementation container (as a VECU file) to be used in ConfigurationDesk.

T

Table A type of pane that offers access to a specific subset of elements and properties of the active ConfigurationDesk application in rows and columns.

TargetLink A dSPACE software product for production code generation. It lets you generate highly efficient C code for microcontrollers in electronic control units (ECUs). It also helps you implement control systems that have been modeled graphically in a Simulink/Stateflow diagram on a production ECU.

TargetLink is able to provide a V-ECU implementation container (as a VECU file) or a Simulink implementation container (SIC file) to be used in ConfigurationDesk.

Task A piece of code whose execution is controlled by a real-time operating system (RTOS). A task is usually triggered by an event, and executes one or more runnable functions. In a ConfigurationDesk application, there are predefined tasks that are provided by executable application components. In addition, you can create user-defined tasks that are triggered, for example, by I/O events. Regardless of the type of task, you can configure the tasks by specifying the priority, the DAQ raster name, etc.

Task Configuration table A pane that lets you configure the tasks of an executable application.

Temporary working view A working view that can be used for drafting a signal chain segment, like a notepad.

Timer event A periodic event with a sample rate and an optional offset.

Topology A hierarchy that serves as a repository for creating a signal chain. All the elements of a topology can be used in the signal chain, but not every element needs to be used. You can export each topology from and import it to a ConfigurationDesk application. Therefore a topology can be used in several applications.

A ConfigurationDesk application can contain the following topologies:

- Device topology
- Hardware topology
- Model topology

TRC file A variable description file that contains all variables (signals and parameters) which can be accessed via the experiment software. It is created during the build process.

TX Communication data that is transmitted by a bus member.

U

User function An external function or program that is added to the Automation – User Functions ribbon group for quick and easy access during work with ConfigurationDesk.

V

VECU file A ZIP file that contains a V-ECU implementation. A VECU file can contain data packages for different platforms. VECU files are exported by TargetLink or SystemDesk. You can add a V-ECU implementation based on a VECU file to the model topology in the same way as adding a Simulink model based on an SLX file.

VEOS The dSPACE software product for performing offline simulation. VEOS is a PC-based simulation platform which allows offline simulation independently from real time.

VEOS Player A software running on the host PC for building offline simulation applications. Offline simulation applications can be downloaded to VEOS to perform offline simulation. ConfigurationDesk lets you generate a bus simulation container (BSC file) via the Bus Manager. You can then import the BSC file into the VEOS Player.

View set A specific arrangement of some of ConfigurationDesk's panes. You can switch between view sets by using the navigation bar. ConfigurationDesk provides preconfigured view sets for specific purposes. You can customize existing view sets and create user-defined view sets.

VSET file A file that contains all view sets and their settings from the current ConfigurationDesk installation. A VSET file can be exported and imported via the View Sets page of the Customize dialog.

W

Working area The central area of ConfigurationDesk's user interface.

Working view A view of the signal chain elements (blocks, ports, mappings, etc.) used in the active ConfigurationDesk application. A working view can be opened in the Signal Chain Browser or the Model Communication Browser. ConfigurationDesk provides two default working views: The Global working view and the Temporary working view. In the Working View Manager, you can also create user-defined working views that let you focus on specific signal chain segments according to your requirements.

Working View Manager A pane that lets you manage the working views of the active ConfigurationDesk application. You can use the Working View Manager for creating, renaming, and deleting working views, and also to open a working view in the Signal Chain Browser or the Model Communication Browser.

X

XLSX file A Microsoft Excel™ file format that is used for the following purposes:

- Creating or configuring a device topology outside of ConfigurationDesk.
- Exporting the wiring information for the external cable harness.
- Exporting the configuration data of the currently active ConfigurationDesk application for documentation purposes.