ConfigurationDesk

# UART Implementation

Release 2021-A – May 2021

dSPACE

## How to Contact dSPACE

| | |
|---|---|
| Mail: | dSPACE GmbH<br>Rathenaustraße 26<br>33102 Paderborn<br>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 7

## Implementing UART Serial Communication in ConfigurationDesk 9

Basics on Implementing a UART Serial Communication Interface.................... 9
Data Flow Between UART Core, UART Driver and UART Custom Function Block........................................................................................ 11

## UART Demo Projects and Applications 13

CfgUARTDemo Project........................................................................................ 14
Example of Simple UART Serial Communication Using a DS2672 Bus Module............................................................................................. 14
Example of UART Serial Communication Using Onboard UART of SCALEXIO Processing Hardware........................................................... 17
Example of UART Serial Communication Using a DS1511 or DS1513 Multi-I/O Board (MicroAutoBox III).............................................................. 19

CfgUARTRS232FlowControlDemo Project.......................................................... 23
Example of Complex UART Serial Communication (RS232 FlowControl) Using a DS2671 Bus Board................................................. 23
Example of UART Serial Communication Using a DS6321 UART Board (SCALEXIO LabBox)........................................................................ 27
Example of UART Serial Communication Using a DS1521 Bus Board (MicroAutoBox III)...................................................................................... 30

Building the Signal Chain for UART Communication Using an FPGA Board............ 33
Example of UART Bus Communication Using an FPGA Application................ 33
Port Description of the FPGA Custom Function Blocks for UART Communication.................................................................................. 35
Building the Demo Project and Experimenting............................................. 40

## ConfigurationDesk UART API Reference 43

DsCIoFuncUart Class.......................................................................................... 44

DsNIoFuncUart Namespace.................................................................................. 44
General Information on the DsNIoFuncUart Namespace................................ 45
DsNIoFuncUart::Baudrate............................................................................... 46

DsNIoFuncUart::ChannelStatus.......... 46
DsNIoFuncUart::ChannelType.......... 47
DsNIoFuncUart::Databits.......... 48
DsNIoFuncUart::ErrorCode.......... 48
DsNIoFuncUart::ErrorStatus.......... 50
DsNIoFuncUart::Event.......... 50
DsNIoFuncUart::FlowControlMode.......... 52
DsNIoFuncUart::Parity.......... 53
DsNIoFuncUart::RxFifoBlockSize.......... 53
DsNIoFuncUart::RxFifoMode.......... 54
DsNIoFuncUart::RxFifoSize.......... 54
DsNIoFuncUart::RxFifoTriggerLevel.......... 55
DsNIoFuncUart::Stopbits.......... 56
DsNIoFuncUart::Termination.......... 56
DsNIoFuncUart::Transceiver.......... 57
DsNIoFuncUart::UartLineStatus.......... 58
DsNIoFuncUart::UartModemStatus.......... 59
DsNStdErrorCode.......... 60

DsCIoFuncUart Methods.......... 60
DsCIoFuncUart::applySettings.......... 62
DsCIoFuncUart::bindIoChannel.......... 63
DsCIoFuncUart::create.......... 66
DsCIoFuncUart::getChannelStatus.......... 67
DsCIoFuncUart::getErrorStatus.......... 68
DsCIoFuncUart::getLineStatus.......... 69
DsCIoFuncUart::getModemStatus.......... 71
DsCIoFuncUart::getRxFifoLevel.......... 72
DsCIoFuncUart::getRxFifoSize.......... 72
DsCIoFuncUart::getTxFifoSpace.......... 73
DsCIoFuncUart::receive.......... 74
DsCIoFuncUart::receiveTerm.......... 76
DsCIoFuncUart::resetRxFifo.......... 77
DsCIoFuncUart::send.......... 78
DsCIoFuncUart::setBaudrate.......... 79
Baud Rates for Onboard UART of SCALEXIO Processing Hardware.......... 80
DsCIoFuncUart::setDatabits.......... 81
DsCIoFuncUart::setFlowControlMode.......... 82
DsCIoFuncUart::setStopbits.......... 83
DsCIoFuncUart::setParity.......... 84
DsCIoFuncUart::setTermination.......... 85
DsCIoFuncUart::setTransceiver.......... 86
DsCIoFuncUart::setTriggerEnable.......... 87

DsCIoFuncUart::setRxFifoBlockSize........ 88
DsCIoFuncUart::setRxFifoMode........ 90
DsCIoFuncUart::setRxFifoSize........ 91
DsCIoFuncUart::setRxFifoTriggerLevel........ 92
DsCIoFuncUart::start........ 93
DsCIoFuncUart::stop........ 94

ConfigurationDesk Glossary 97

Index 123

# About This Document

**Content**

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). 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.

**Required knowledge**

You should be familiar with:

- Basics of ConfigurationDesk.
- UART serial communication.
- The C++ programming language and the XML language.

**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 UART Serial Communication in ConfigurationDesk

**Where to go from here**

Information in this section

Basics on Implementing a UART Serial Communication Interface..............9

Data Flow Between UART Core, UART Driver and UART Custom Function Block.........................................................................11

## Basics on Implementing a UART Serial Communication Interface

**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 a description of the API for the UART driver, refer to ConfigurationDesk UART API Reference on page 43.

**Custom function block for implementing a UART interface**

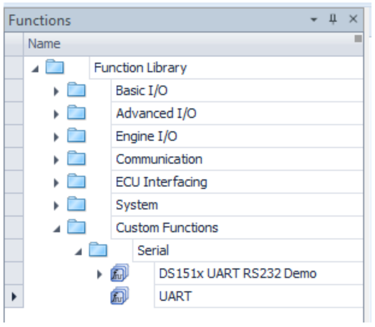
ConfigurationDesk has no standard function block for implementing a UART interface. You must use a custom function block for this purpose. A custom function block consists of a C++ source code module and a function block. In the C++ source code module a driver object must be implemented which drives the channel of a bus board or module. The function block is required to build the signal chain in ConfigurationDesk, to assign a hardware resource to the driver object and finally to integrate the driver object into the real-time application. For more information on creatingcustom function blocks, refer to ConfigurationDesk Custom I/O Function Implementation Guide 🕮.

**Note**

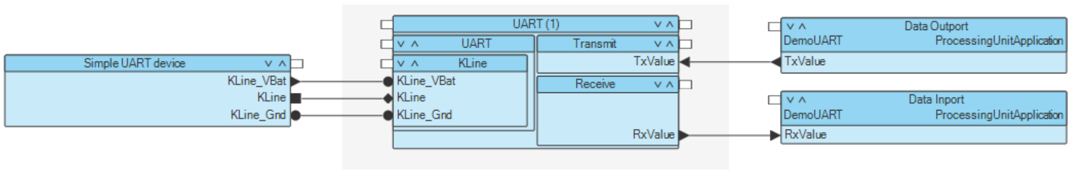
Simulink models used in the ConfigurationDesk application must not have the same name as any of the custom function files in the custom function directories. For example, you must not use `UART.mdl` as a model name because **UART** is the name of a custom function transformed to a valid C identifier.

**Building the signal chain**

Custom function block types are available in folders in the **Function Browser** like standard function blocks. The following illustration shows the custom function block types in the **CfgUARTDemo** project.



Custom function blocks can be used in a signal chain in the same way as standard function blocks. For details, refer to Implementing I/O Functionality (ConfigurationDesk Real-Time Implementation Guide 🕮). The following illustration shows a signal chain that uses a **UART** custom function block.



The bus line to the UART can be used in a failure simulation. You can set the allowed failure classes in the same way as for standard function blocks, refer to Specifying Failure Simulation in ConfigurationDesk (ConfigurationDesk Real-Time Implementation Guide 🕮).

**Examples in demo projects**

There are several examples of UART implementations included in the ConfigurationDesk demo projects. They demonstrate how a custom function block can be used in the signal chain to create a serial communication interface. You can use an example as a basis for your own implementation.

The following UART demo projects and applications are available:

| Project | Application | Short Description | Refer to... |
|---|---|---|---|
| CfgUARTDemo | UARTAppl | This application demonstrates a UART implementation for the DS2672 Bus Module of a DS2680 I/O Unit. This module has a LIN/K-Line transceiver which can be used for serial communication. | Example of Simple UART Serial Communication Using a DS2672 Bus Module on page 14<br>**Tip**<br>You can modify the **UARTAppl** application to use the demo with onboard UART of SCALEXIO processing hardware. Refer to Example of UART Serial Communication Using Onboard UART of SCALEXIO Processing Hardware on page 17. |
| | UARTRS232_MABXIIIAppl | This application demonstrates a UART implementation for the DS1511 Multi-I/O Board of a MicroAutoBox III. | Example of UART Serial Communication Using a DS1511 or DS1513 Multi-I/O Board (MicroAutoBox III) on page 19 |
| CfgUARTRS232FlowControlDemo | UARTRS232FlowControlAppl | This application demonstrates a UART implementation for a DS2671 Bus Board. | Example of Complex UART Serial Communication (RS232 FlowControl) Using a DS2671 Bus Board on page 23 |
| | DS6321_UARTAppl | This application demonstrates a UART implementation for the DS6321 UART Board of a SCALEXIO LabBox. | Example of UART Serial Communication Using a DS6321 UART Board (SCALEXIO LabBox) on page 27 |
| | DS1521_MABXIII_UARTAppl | This application demonstrates a UART implementation for the DS1521 Bus Board of a MicroAutoBox III. | Example of UART Serial Communication Using a DS1521 Bus Board (MicroAutoBox III) on page 30 |
| CfgFPGAuartDemo | FPGAuartDemo | This application demonstrates a UART implementation for the DS2655M2 Digital I/O Module of a DS2655 (7K160) FPGA Base Board. | Building the Signal Chain for UART Communication Using an FPGA Board on page 33 |

## Data Flow Between UART Core, UART Driver and UART Custom Function Block

**Hardware and driver for serial communication**

**UART core** The UART core is a hardware unit of a dSPACE system that realizes UART support at the lowest level, for example, sending and receiving bytes via the connected transceiver, error and status handling, event generation.

**UART driver** A UART driver is a software layer above the UART core. It concentrates the basic UART core functionality as simple manageable methods and provides them in a C++ API. The UART driver has a software FIFO for receive data that you can easily use to read data and status information.

**Polling data in periodical tasks**

A latency of one sample step can occur when data is polled in periodical tasks. If a UART driver is used in a periodical task, the UART driver reads the received data from the UART core at the beginning of the task and sends it to the computation node. The UART driver of the computation node does not wait until all data has been received for performance reasons. The UART driver API can therefore access only data that is received at the time when the data is read in the task at the computation node. Depending on the structure and run time of the executable application, not all data may be available for the user code within one sample step. A latency of one sample step can occur. To avoid this latency, you can use I/O events.

**Triggering I/O events from the UART core**

You can start tasks and their functions not only periodically but also asynchronously via I/O events. The UART core is one way to generate I/O events. Select the corresponding instance of the UART driver as the hardware resource for an I/O event (see Defining I/O Events (Serial Communication) (ConfigurationDesk Custom I/O Function Implementation Guide 🕮)). In ConfigurationDesk, you can assign the I/O event to a task during task configuration (for details, refer to Introduction to Modeling Executable Applications and Tasks (ConfigurationDesk Real-Time Implementation Guide 🕮)). In addition, the UART core must be configured using the C++ API of the UART driver so that events can be generated and transmitted.

If the UART driver was configured for event triggering and the relevant event occurs, all data which is received in the UART core is transferred to the UART driver, regardless of whether the event triggers a task or not. This ensures that all received data is available in the user code, which is not the case if you poll for the data.

**Binding UART driver to several functions**

Instances of a UART driver can be bound to several functions. The functions can be executed in different periodic or asynchronous tasks. Note the following points:

- A UART driver has only one FIFO for received data. Data which is read by one function is invisible to the other functions.
- Send data which is written by different functions is sent sequentially.

# UART Demo Projects and Applications

**Where to go from here**

Information in this section

CfgUARTDemo Project....14
CfgUARTRS232FlowControlDemo Project....23
Building the Signal Chain for UART Communication Using an FPGA Board....33

## CfgUARTDemo Project

**Where to go from here**

Information in this section

Example of Simple UART Serial Communication Using a DS2672 Bus Module........ 14
Example of UART Serial Communication Using Onboard UART of SCALEXIO Processing Hardware........ 17
Example of UART Serial Communication Using a DS1511 or DS1513 Multi-I/O Board (MicroAutoBox III)........ 19

## Example of Simple UART Serial Communication Using a DS2672 Bus Module

**Demo project**

The UART custom function block in the UARTAppl application of the CfgUARTDemo demo project is implemented for a DS2672 Bus Module of a DS2680 I/O Unit. This module has a LIN/K-Line transceiver which can be used for serial communication.

The UARTAppl application is a simple example of serial communication, prepared for your first experience with custom function blocks. It shows

- Basics of custom function blocks
- Initializing a UART driver
- Using configuration properties
- Start/stop functions
- Sending and receiving data of a fixed length

**UART custom function block**

Four file types are necessary for a custom function block. An XML file containing all the descriptions for ConfigurationDesk, two header files (.h file) containing the function declarations and type definitions, and a C++ source code file (.cpp file) containing the function definitions. You can find the files of the demo UART custom function block in the `UARTDemo\CfgUARTDemo\CustomFunctions` project folder in your Documents folder.

The custom function block types are displayed in the Function Browser under the Custom Functions folder.

The UART function block is generated based on the `UART.xml` file. This file references the `UART.cpp`. To get information on the implementation, examine these files and refer to Example: Implementing a Custom Function Block for UART Serial Communication (ConfigurationDesk Custom I/O Function Implementation Guide).

**Hardware preconditions**

If you want to use the implementation on a hardware platform that is available, the following preconditions must be met.

**Tip**

You do not require the actual hardware to implement and build the real-time application in ConfigurationDesk.

**Hardware requirement** The following hardware is required if you want to use the demo scenario with the UART custom function block:

- A SCALEXIO system with a DS2680 I/O Unit and a DS2672 Bus Module is required. The bus module has two channels of the LIN 1 channel type. This channel type has a LIN/K-Line transceiver which can be used for UART-based communication.
- A power supply (up to 35 V) for the LIN/K-Line transceiver is required. You can use an external power supply or the battery voltage simulation of the SCALEXIO system (see Connecting External Devices to the Power Supply (SCALEXIO – Hardware and Software Overview 🕮)).

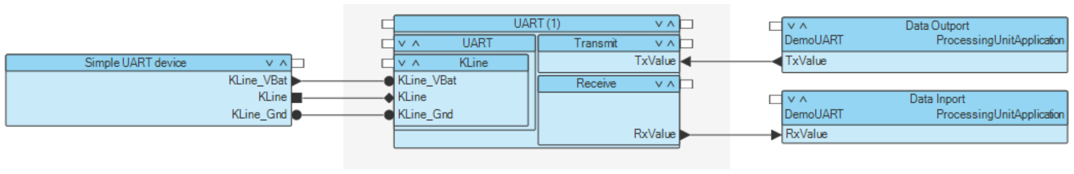
**Hardware connection** To run the demo, the following pins must be connected.

| Signal | Pin | Description |
|---|---|---|
| VBAT (LIN) | ECU2 connector, E 7 pin | Battery supply voltage for the LIN/K-Line transceiver (6 V ... 35 V) |
| GND (Bus) | ECU2 connector, B 9 pin | Ground for the LIN/K-Line transceiver |
| LIN 1 Channel 1 Signal<br>or<br>LIN 1 Channel 2 Signal | ECU2 connector, E 8 pin<br>or<br>ECU2 connector, E 6 pin | Bus signal. If you only want to execute the UART_Simple demo, it is not necessary to connect the pin. |

When the ConfigurationDesk application is loaded and the hardware assignment is done, you can also get the pin numbers of the function block via the Properties Browser. For details, refer to Configuring Function Blocks (ConfigurationDesk Real-Time Implementation Guide 🕮).

**Signal chain**

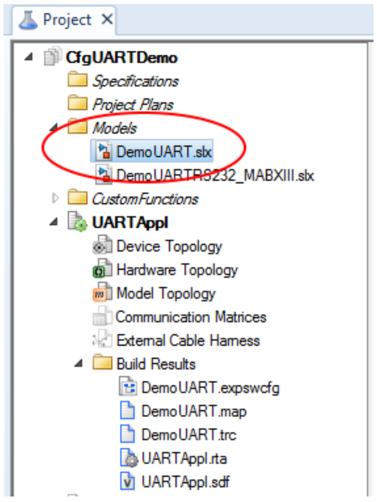
The following illustration shows the signal chain with the UART function block.



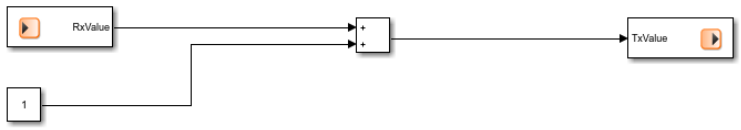
The TxValue value of the model is written to the Transmit function of the UART function block. The function sends the UInt32 value in packages of 4 bytes via the LIN/K-Line transceiver. The external device in the left column of the signal chain is optional. It is not required to execute the demo. The LIN/K-Line transceiver sends and receives on the same bus line. An external connection is therefore not required. The **Receive** function of the **UART** function block receives the bytes and collects them to a UInt32 value. Afterwards, the value is written to the model.

**Simulink model**

When you open the **CfgUARTDemo** project, the Simulink model of the demo is available in the **Models** folder in the **Project** view set.



You can open the model in Simulink.



The model is very simple. A UInt32 value is read from the **RxValue** Inport. A value of 1 is added to the read value and the result is written to the **TxValue** outport.

**ControlDesk experiment**

A ControlDesk project with an experiment using the **UART** function block is available in in the `UARTDemo\CDNG_UARTDemo` folder in the Documents folder of your ConfigurationDesk installation.

In ControlDesk, you can start the real-time application and observe the counter value. For details on working with ControlDesk, refer to ControlDesk Introduction and Overview .

## Example of UART Serial Communication Using Onboard UART of SCALEXIO Processing Hardware

**Introduction**

SCALEXIO processing hardware components provide a UART channel that you can use to implement UART serial communication. You can modify the **UARTAppl** application in the **CfgUARTDemo** project for simple UART serial communication to work with the UART channels provided by the processing hardware.

For a description of the **UARTAppl** application, refer to Example of Simple UART Serial Communication Using a DS2672 Bus Module on page 14.

**Hardware preconditions**

**Hardware requirements** You require one of the following hardware components:

- A SCALEXIO Processing Unit provides a single channel of the UART 5 channel type.
- The DS6001 Processor Board provides a single channel of the UART 6 channel type.

**Hardware connection** To use the demo, the RX pin (2) and the TX pin (3) of the UART RS232 connector must be connected.

**Limitations**

There are several limitations when implementing UART serial communication with the onboard UART of SCALEXIO processing hardware:

- Do not use the SCALEXIO processing hardware UART communication for high-performance data applications.The transmit and receive FIFOs are only 16 bytes in size (as opposed to 1024 bytes for the DS2671 Bus Board, for example).
- Do not use the SCALEXIO processing hardware UART in a model that requires high and accurate clock rates.The UART communication is not operated by a separate process. Instead, it is processed in parallel to the model tasks on the processing hardware. During the driver access to the UART, the interrupts on the processing hardware are disabled. This can have a temporal effect on the model behavior.
- UART on SCALEXIO processing hardware operates with a fixed frequency. Only the divisor in the UART can be changed to set the baud rate. Due to quantization effects, only rough steps are possible, especially in the high baud rate range. For more information on calculating and setting supported baud rates, refer to Baud Rates for Onboard UART of SCALEXIO Processing Hardware on page 80.

- RX interrupts are not only triggered when the set trigger level in the RX-FIFO is reached. An RX interrupt is also triggered when data is in the RX-FIFO and nothing has been received for the period of 4 character times (timeout) before the trigger level is reached. A character time is the time required to receive 4 bytes. The timeout interrupt is not configurable. This offers the advantage that incompletely received data no longer remains in the FIFO. However, on an RX interrupt you cannot rely on the amount of data set as trigger level to be available when reading out the data. There may be fewer bytes if the interrupt was triggered by the timeout. The number of received bytes is returned by the read function.
- The UART of the SCALEXIO Processing Unit does not support automatic flow control.

## Modifying the CfgUARTDemo project

To modify the UARTAppl application to work with the UART channel provided by SCALEXIO processing hardware, apply the following steps:

1. Open the UARTAppl application and use the Save Project As command on the File – Save As ribbon to save the CfgUARTDemo project under a new name.
2. From the `CustomFunctions` folder of the project, open the `UART.xml` file with a text or XML editor.
3. Increment the version and change the distributor information of the custom function block type (bold print):

```
<CustomFunctionBlock Version="2" Name="UART" DistributorInformation="dSPACE GmbH">
```

4. Apply the following changes (bold print) to the logical signals:

```
<LogicalSignals>
  <LogicalSignal Name="RS232" Id="LogSig1">
    <HwRequirements>
      <Resource Name="Data_Res" Id="Resource1" ResourceType="UART_5" />
    </HwRequirements>
    <SignalPorts>
      <SignalPort Name="TX" Id="SP1" Direction="Out" IsFiuEnabled="false">
        <ResourceSignal Name="UartTx" ResourceId="Resource1" />
      </SignalPort>
      <SignalPort Name="RX" Id="SP2" Direction="In" IsFiuEnabled="false">
        <ResourceSignal Name="UartRx" ResourceId="Resource1" />
      </SignalPort>
      <SignalPort Name="GND" Id="SP3" Direction="Reference" IsFiuEnabled="false">
        <ResourceSignal Name="UartGnd" ResourceId="Resource1" />
      </SignalPort>
    </SignalPorts>
  </LogicalSignal>
```

   Here, the `ResourceType` `UART_5` for the SCALEXIO Processing Unit is used. For the DS6001 Processor Board use `UART_6`.

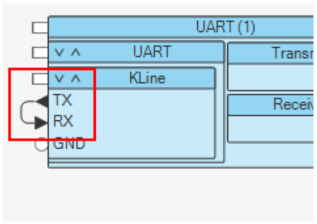
5. Save and close the `UART.xml` file.
6. From the `CustomFunctions` folder of the project, open the `UART.cpp` file and apply the following change (bold print) in the `ucf_Uart_OneTimeInit()` function to change the transceiver type:

   ```
   pUartDrv->setTransceiver(ErrorList,DsNIoFuncUart::Transceiver::RS232);
   ```

7. Save and close the `UART.cpp` file.

8. In ConfigurationDesk, open the **Signal Chain** view set.
9. In the **Function Browser**, right-click the **UART** function block type and select **Reload Definition for 'UART'**.

   ConfigurationDesk updates the **UART (1)** custom function block instance in the signal chain to the new version.
10. Delete the **Simple UART device** from the ConfigurationDesk application.
11. Map the **TX** signal port to the **RX** signal port.



12. Right-click the **UART (1)** function block and select **Hardware Assignment – Assign Channel Set – UART 5 [...]**.
13. On the **Home – Build** ribbon, click **Start** to start the build process.

    Wait until the build process is finished. This may take a couple of minutes.

**Modifying the ControlDesk demo project**

To observe the real-time application in ControlDesk, you can use a modified version of the **CD_UARTDemo** project.

Apply the following steps:

1. In ControlDesk, open the CD_UartDemo project.
2. Select **Replace** from the context menu of the **UARTAppl.sdf** file in the **UARTExp** experiment.
3. Select the SDF file from the `Build Results` folder of the modified ConfigurationDesk demo project and click **Open**.

   You can now start measuring on a connected platform.

## Example of UART Serial Communication Using a DS1511 or DS1513 Multi-I/O Board (MicroAutoBox III)

**Demo project**

The **DS151x UART RS232 Demo** custom function block in the **UARTRS232_MABXIIIAppl** application of the **CfgUARTDemo** demo project is implemented for a DS1511 or DS1513 Multi-I/O Board of a MicroAutoBox III.

**UART custom function block**

Four file types are necessary for a custom function block. An XML file containing all the descriptions for ConfigurationDesk, two header files (.h file) containing the function declarations and type definitions, and a C++ source code file (.cpp file) containing the function definitions. You can find the files of the demo **DS151x UART RS232 Demo** custom function block in the

`UARTDemo\CfgUARTDemo\CustomFunctions` project folder in your Documents folder.

The custom function block type is displayed in the **Function Browser** under the **Custom Functions** folder.

The **DS151x UART RS232 Demo** function block is generated based on the `DS151xUARTDemo.xml` file. This file references the `DS151xUARTDemo.cpp`. For general information on the implementation of custom function blocks, refer to ConfigurationDesk Custom I/O Function Implementation Guide.

## Hardware preconditions

If you want to use the implementation on a hardware platform that is available, the following preconditions must be met.

**Tip**

You do not require the actual hardware to implement and build the real-time application in ConfigurationDesk.

**Hardware requirement** The following hardware is required if you want to use the demo scenario with the **DS151x UART RS232 Demo** custom function block:

- A MicroAutoBox III with a DS1511 or DS1513 Multi-I/O Board is required. The boards have channels of the UART 2 channel type. This channel type supports the RS232 transceiver type.

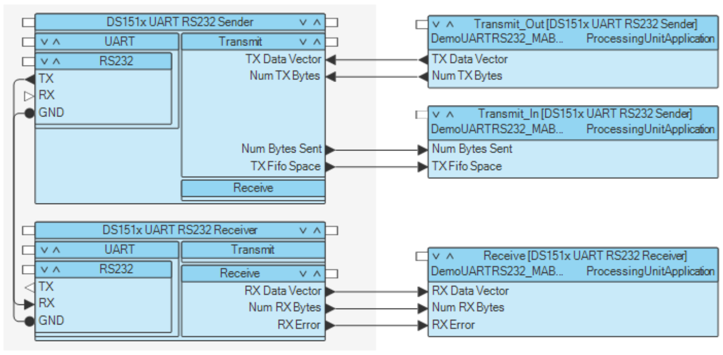
**Hardware connection** To run the demo, the following pins must be connected (example for the DS1511 Multi I/O Board).

| Signal | Pin | Description |
|---|---|---|
| UartTX1 | c5 pin | Transmit signal of the CAN Type 1 (Module 1) channel. |
| UartRX1 | c6 pin | Receive signal of the CAN Type 1 (Module 1) channel. |
| UartTX2 | B5 pin | Transmit signal of the CAN Type 1 (Module 2) channel. |
| UartRX2 | B6 pin | Receive signal of the CAN Type 1 (Module 2) channel. |
| GND | – | The ground pins of the DS1511 Multi I/O Board are internally connected. |

When the ConfigurationDesk application is loaded and the hardware assignment is done, you can also get the pin numbers of the function block via the **Properties Browser**. For details, refer to Configuring Function Blocks (ConfigurationDesk Real-Time Implementation Guide).

**Signal chain**

The following illustration shows the signal chain with the DS151x UART RS232 Demo function blocks.



The DS151x UART RS232 Sender and DS151x UART RS232 Receiver function blocks are instances of the DS151x UART RS232 Demo custom function block type.

The demo offers a number of in- and outports to send and receive data:

- TX Data Vector inport: A vector with a data width of 63.
- Num TX Bytes inport: The number of bytes to be sent by TX Data Vector.
- Num Bytes Sent outport: The number of bytes that were actually sent by TX Data Vector.
- TX Fifo Space outport: Shows the number of bytes that can be written to the TX-FIFO with no data loss at the current simulation step.
- RX Data Vector outport: A vector with a data width of 63. The received bytes of data are output here.
- Num RX Bytes outport: The number of bytes received by RX Data Vector. Only that number of bytes is valid and no further bytes are to be processed.
- RX Error outport: 0 means that no error was detected.

  If an error was detected, a bit field based on the line status register of the UART is received. The following errors can be detected:

  - (0x02) Overrun Error: An overrun occured to the hardware receive FIFO. Data was lost.
  - (0x04) Parity Error: A byte was received with a parity error (Data specific bit).
  - (0x08) Framing Error: A byte was received with a framing error (Data specific bit).
  - (0x10) Break Interrupt: A break interrupt was detected (Trigger-bit).
  - (0x80) Error in receive FIFO: At least one Overrun, Parity, or Framing Error or Break Interrupt in receive FIFO.

**Debug messages** The Transmit function and the Receive function of the function blocks offer properties to enable debug messages. The debug messages are written to the dSPACE Log and to `http://<MicroAutoBoxIII_IP_address>/action/messages`.

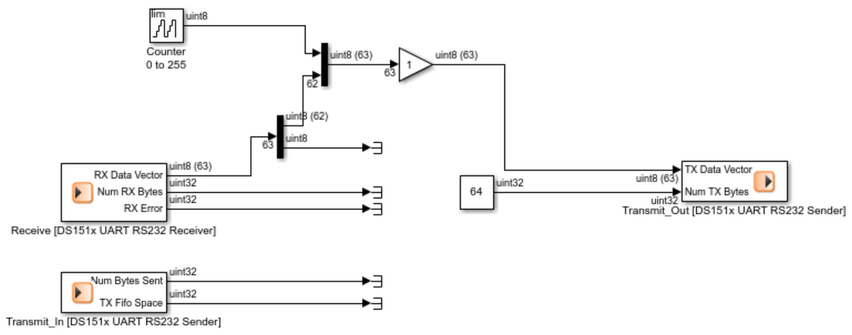
The following properties for debug messages are available:

- TX Data Info: Shows the first bytes from TX Data Vector.
- RX Data Info: Shows the first bytes from RX Data Vector.
- RX Error Info: Shows errors that were detected.

## Simulink model

When you open the CfgUARTDemo project, the Simulink model of the demo is available in the Models folder in the Project view set.

You can open the model in Simulink.



## Related topics

References

UART 2 Characteristics (MicroAutoBox III Hardware Installation and Configuration 🕮)
UART 2 Characteristics (MicroAutoBox III Hardware Installation and Configuration 🕮)

# CfgUARTRS232FlowControlDemo Project

**Where to go from here**

Information in this section

Example of Complex UART Serial Communication (RS232 FlowControl) Using a DS2671 Bus Board....................................... 23

Example of UART Serial Communication Using a DS6321 UART Board (SCALEXIO LabBox)............................................................. 27

Example of UART Serial Communication Using a DS1521 Bus Board (MicroAutoBox III)............................................................. 30

## Example of Complex UART Serial Communication (RS232 FlowControl) Using a DS2671 Bus Board

**Demo project**

The UART RS232 FlowControl custom function block in the UARTRS232FlowControlAppl application of the CfgUARTRS232FlowControlDemo demo project is implemented for a DS2671 Bus Board.

The UARTRS232FlowControlAppl application is a complex example of serial communication, prepared for your advanced experience with custom function blocks. It shows

- Sending and receiving data of arbitrary length
- Usage of several hardware resources
- Generating events

**UART RS232 FlowControl custom function block**

Four file types are necessary for a custom function block. An XML file containing all the descriptions for ConfigurationDesk, two header files (.h file) containing the function declarations and type definitions, and a C++ source code file (.cpp file) containing the function definitions. You can find the files of the demo UART RS232 FlowControl custom function block in the `UARTDemo\CfgUARTRS232FlowControlDemo\CustomFunctions` project folder in your Documents folder.

The custom function block type is displayed in the Function Browser under the Custom Functions folder.

The UART RS232 FlowControl custom function block is generated based on the `UART_RS232_FlowControl.xml` file. This file references the `UART_RS232_FlowControl.cpp`. To get information on the implementation, examine these files and refer to Example: Implementing a Custom Function Block for UART Serial Communication (ConfigurationDesk Custom I/O Function Implementation Guide 🕮).

**Hardware preconditions**

If you want to use the implementation on a hardware platform that is available, the following preconditions must be met.

**Tip**

You do not require the actual hardware to implement and build the real-time application in ConfigurationDesk.

**Hardware requirement** The following hardware is required if you want to use the demo scenario with the **UART RS232 FlowControl** function block:

- A SCALEXIO system with a DS2671 Bus Board is required. The board has four channels of the **Bus 1** channel type. This channel type has several transceiver types for each channel. One transceiver type is the RS232 transceiver which can be used for UART-based communication. One function block requires two consecutive channels. As a sender and a receiver is configured in the demo, the demo uses all four channels of a DS2671.

**Hardware connection** A sender and a receiver is configured in the demo. To run the demo, signal and reference lines of the sender must be connected to the receiver, see the following table.

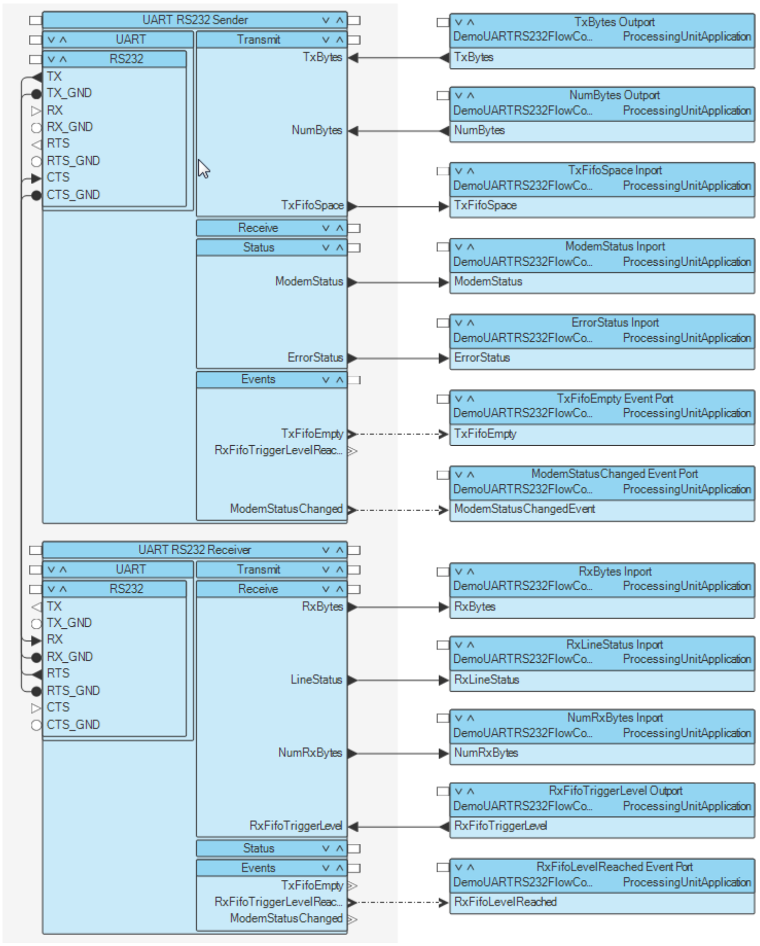
| **Sender** | | | | **Receiver** | | |
|---|---|---|---|---|---|---|
| **Signal** | **Channel** | **Pin Name** | | **Signal** | **Channel** | **Pin Name** |
| TX | 1 | PinA | <---> | RX | 3 | PinC |
| TX_GND | 1 | PinB | <---> | RX_GND | 3 | PinD |
| CTS | 2 | PinC | <---> | RTS | 4 | PinA |
| CTS_GND | 2 | PinD | <---> | RTS_GND | 4 | PinB |

It is not necessary to connect the remaining pins (RX, RX_GND, RTS, RTS_GND of the sender and TX, TX_GND, CTS, CTS_GND of the receiver).

The pins that must be connected depend on the hardware configuration (for example, the slot which is used for the DS2671 Bus Board) and the hardware assignments (the channel numbers that are assigned to the function blocks). When the ConfigurationDesk application is loaded and the hardware assignment is done, you can get the pin numbers of the function block via the **Properties Browser**. For details, refer to Configuring Function Blocks (ConfigurationDesk Real-Time Implementation Guide 🕮).

## Signal chain

The following illustration shows the signal chain with the UART RS232 FlowControl function blocks.

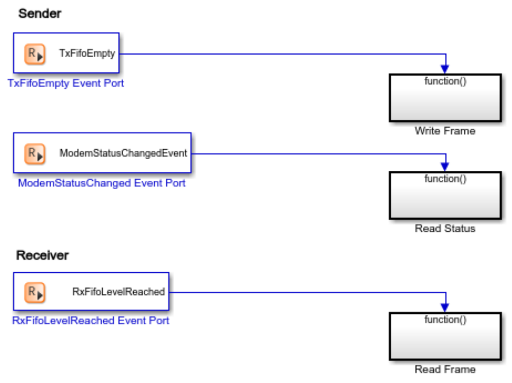


The UART RS232 Sender and UART RS232 Receiver function blocks are instances of the UART RS232 FlowControl custom function block type.

## Simulink model

When you open the CfgUARTRS232FlowControlDemo project, the Simulink model of the demo is available in the Models folder in the Project view set.

The following illustration shows the root level of the Simulink model.



The demo consists of three subsystems which are called when events are triggered.

**Write Frame subsystem** The Write Frame subsystem is triggered when the transmit FIFO is empty.

The subsystem sends a frame with a counter on the first byte. All other bytes are zero. The counter is incremented by each time the subsystem is executed. The NumBytes constant is used to set the frame length. The value can be changed in the experiment. The TxFifoSpace inport provides the minimum number of bytes that fit into the transmit FIFO of the channel.

**Read Status subsystem** The Read Status subsystem is triggered when the modem status has changed.

The Modem Status and Error Status inports provide information on the modem and error status.

**Read Frame subsystem** The Read Frame subsystem is triggered when the selected receive FIFO level was reached.

The RxBytes inport provides the values of the received bytes. The RxLineStatus inport provides the line status of the receiver. The NumRxBytes inport provides the number of received bytes. The RxFifoTriggerLevel constant is used to set the trigger level. The value can be changed in the experiment.

**ControlDesk experiment**

A ControlDesk project with an experiment using UART RS232 FlowControl function blocks is available in the `UARTDemo\CDNG_UARTRS232FlowControlDemo` folder in the Documents folder of your ConfigurationDesk installation.

In ControlDesk, you can start the real-time application and observe the counter value. You can set properties for the sender and receiver. For details on working with ControlDesk, refer to ControlDesk Introduction and Overview.

You can simulate a malfunction: If you open the connection between RTS and CTS, the data transfer is interrupted. The modem status is changed and the counter value is not incremented any longer.

## Example of UART Serial Communication Using a DS6321 UART Board (SCALEXIO LabBox)

**Demo project**

The **DS6321 UART Demo** custom function block in the **DS6321_UARTAppl** application of the **CfgUARTRS232FlowControlDemo** demo project is implemented for a DS6321 UART Board of a SCALEXIO LabBox. This board offers different transceiver types which can be used for serial communication (K-Line, RS232, RS422, RS485). The demo application is configured for the K-Line transceiver.

**UART custom function block**

Four file types are necessary for a custom function block. An XML file containing all the descriptions for ConfigurationDesk, two header files (.h file) containing the function declarations and type definitions, and a C++ source code file (.cpp file) containing the function definitions. You can find the files of the demo **DS6321 UART Demo** custom function block in the `UARTDemo\CfgUARTRS232FlowControlDemo\CustomFunctions` project folder in your Documents folder.

The custom function block type is displayed in the **Function Browser** under the **Custom Functions** folder.

The **DS6321 UART Demo** function block is generated based on the `DS6321_UART.xml` file. This file references the `DS6321_UART.cpp`. For general information on the implementation of custom function blocks, refer to ConfigurationDesk Custom I/O Function Implementation Guide.

**Hardware preconditions**

If you want to use the implementation on a hardware platform that is available, the following preconditions must be met.

**Tip**

You do not require the actual hardware to implement and build the real-time application in ConfigurationDesk.

**Hardware requirement** The following hardware is required if you want to use the demo scenario with the **DS6321 UART Demo** custom function block:

- A SCALEXIO LabBox system with a DS6321 UART Board is required. The board has four channels of the UART 1 channel type. This channel type supports the following transceiver types:
  - K-Line (default configuration in the demo application)
  - RS232
  - RS422
  - RS485

The **Transceiver** property of the function block lets you configure the transceiver type. For K-Line, RS422, and RS485, you can enable the **Termination** property. RS232 and RS422 support RTS/CTS flow control, which you can configure via the **Flow Control Mode** property.

**Note**

If you select a different transceiver, additional adjustments have to be made to the application or the signal pin mapping is not correct.

- A power supply (up to 12 V) for the K-Line transceiver is required. You can use an external power supply or the battery voltage simulation of the SCALEXIO system (see Connecting External Devices to the Power Supply (SCALEXIO – Hardware and Software Overview 📖)).

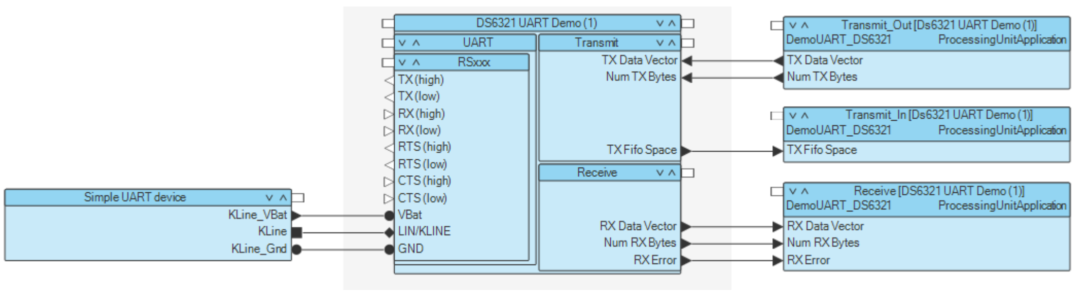
**Hardware connection** To run the demo, the following pins must be connected.

| Signal | Pin | Description |
|---|---|---|
| VBAT | SUBD 34 | Battery supply voltage for the LIN/K-Line transceiver (6 V ... 12 V) |
| GND | SUBD 1 | Ground for the LIN/K-Line transceiver |
| LIN/KLINE | SUBD 36 | Bus signal. If you only want to execute the DS6321 UART demo, it is not necessary to connect the pin. |

When the ConfigurationDesk application is loaded and the hardware assignment is done, you can also get the pin numbers of the function block via the **Properties Browser**. For details, refer to Configuring Function Blocks (ConfigurationDesk Real-Time Implementation Guide 📖).

## Signal chain

The following illustration shows the signal chain with the **DS6321 UART Demo** function block.



The external device in the left column of the signal chain is optional. It is not required to execute the demo.

The demo offers a number of in- and outports to send and receive data:

- TX Data Vector inport: A vector with a data width of 64.
- Num TX Bytes inport: The number of bytes to be sent by TX Data Vector.
- TX Fifo Space outport: Shows the number of bytes that can be written to the TX-FIFO with no data loss at the current simulation step.
- RX Data Vector outport: A vector with a data width of 64. The received bytes of data are output here.
- Num RX Bytes outport: The number of bytes received by RX Data Vector. Only that number of bytes is valid and no further bytes are to be processed.
- RX Error outport: 0 means that no error was detected.

  If an error was detected, a bit field based on the line status register of the UART is received. The following errors can be detected:

  - (0x02) Overrun Error: An overrun occured to the hardware receive FIFO. Data was lost.
  - (0x04) Parity Error: A byte was received with a parity error (Data specific bit).
  - (0x08) Framing Error: A byte was received with a framing error (Data specific bit).
  - (0x10) Break Interrupt: A break interrupt was detected (Trigger-bit).
  - (0x80) Error in receive FIFO: At least one Overrun, Parity, or Framing Error or Break Interrupt in receive FIFO.

**Debug messages** The Transmit and Receive functions offer properties to enable debug messages. The debug messages are written to the dSPACE Log and to `http://<SCALEXIO_IP_address>/action/messages`.

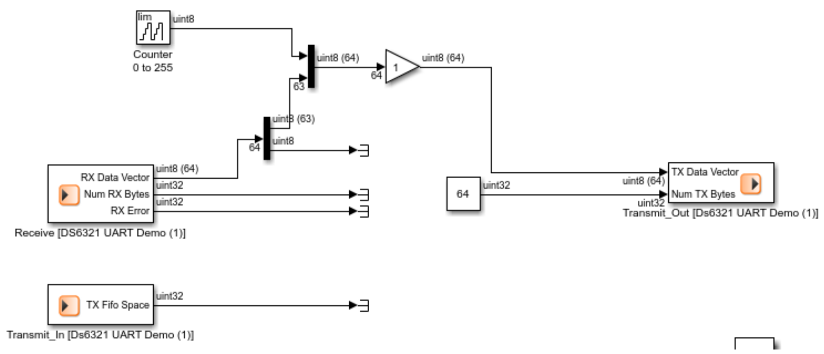
The following properties for debug messages are available:

- TX Data Debug Info: Shows the first bytes from TX Data Vector.
- RX Data Debug Info: Shows the first bytes from RX Data Vector.
- RX Error Debug Info: Shows errors that were detected.

**Simulink model**

When you open the CfgUARTRS232FlowControlDemo project, the Simulink model of the demo is available in the Models folder in the Project view set.

You can open the model in Simulink.



## Example of UART Serial Communication Using a DS1521 Bus Board (MicroAutoBox III)

**Demo project**

The **DS1521 UART Demo** custom function block in the **DS1521_MABXIII_UARTAppl** application of the **CfgUARTRS232FlowControlDemo** demo project is implemented for a DS1521 Bus Board of a MicroAutoBox III. This board offers different transceiver types which can be used for serial communication (RS232, RS422, RS485). The demo application is configured for the RS232 transceiver.

**UART custom function block**

Four file types are necessary for a custom function block. An XML file containing all the descriptions for ConfigurationDesk, two header files (.h file) containing the function declarations and type definitions, and a C++ source code file (.cpp file) containing the function definitions. You can find the files of the demo **DS1521 UART Demo** custom function block in the `UARTDemo\CfgUARTRS232FlowControlDemo\CustomFunctions` project folder in your Documents folder.

The custom function block type is displayed in the **Function Browser** under the **Custom Functions** folder.

The **DS1521 UART Demo** function block is generated based on the `DS1521_UART.xml` file. This file references the `DS1521_UART.cpp`. For general information on the implementation of custom function blocks, refer to ConfigurationDesk Custom I/O Function Implementation Guide.

**Hardware preconditions**

If you want to use the implementation on a hardware platform that is available, the following preconditions must be met.

**Tip**

You do not require the actual hardware to implement and build the real-time application in ConfigurationDesk.

**Hardware requirement** A MicroAutoBox III with a DS1521 Bus Board is required if you want to use the demo scenario with the **DS1521 UART Demo** custom function block. The board has one channel of the UART 4 channel type. This channel type supports the following transceiver types:

- RS232
- RS422
- RS485

The **Transceiver** property of the function block lets you configure the transceiver type. For RS422 and RS485, you can enable the **Termination** property. RS232 supports RTS/CTS flow control, which you can configure via the **Flow Control Mode** property.

**Note**

If you select a different transceiver, additional adjustments have to be made to the application or the signal pin mapping is not correct. Refer to UART 4 Characteristics (MicroAutoBox III Hardware Installation and Configuration 🕮).

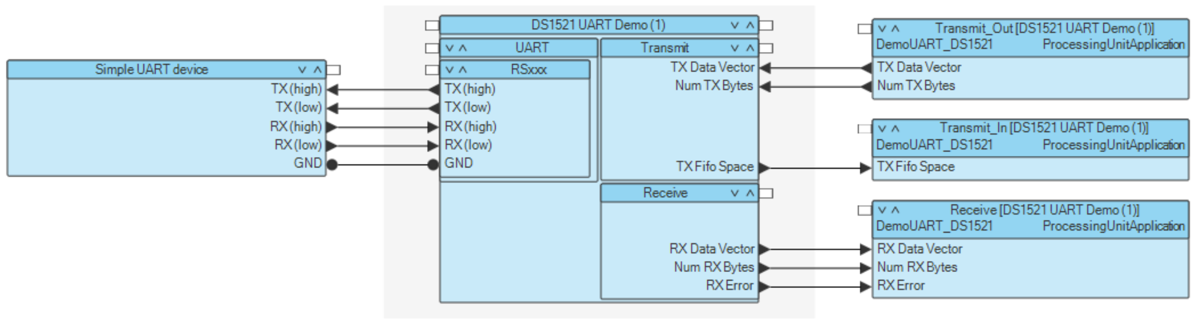
**Hardware connection** To run the demo, the following pins must be connected.

| Signal | Pin | Description |
|---|---|---|
| UART 4 Serial TX- | 10 | Serial transmit signal (TX) |
| UART 4 Serial TX+ | 11 | Request to send (RTS) |
| UART 4 Serial RX+ | 12 | Clear to send (CTS) |
| UART 4 Serial RX- | 13 | Serial receive signal (RX) |
| GND | 14,15,16,17,22,23,24,25 | Common ground |

When the ConfigurationDesk application is loaded and the hardware assignment is done, you can also get the pin numbers of the function block via the Properties Browser. For details, refer to Configuring Function Blocks (ConfigurationDesk Real-Time Implementation Guide 🕮).

## Signal chain

The following illustration shows the signal chain with the DS1521 UART Demo function block.



The external device in the left column of the signal chain is optional. It is not required to execute the demo.

The demo offers a number of in- and outports to send and receive data:

- TX Data Vector inport: A vector with a data width of 64.
- Num TX Bytes inport: The number of bytes to be sent by TX Data Vector.
- TX Fifo Space outport: Shows the number of bytes that can be written to the TX-FIFO with no data loss at the current simulation step.
- RX Data Vector outport: A vector with a data width of 64. The received bytes of data are output here.
- Num RX Bytes outport: The number of bytes received by RX Data Vector. Only that number of bytes is valid and no further bytes are to be processed.

- RX Error outport: 0 means that no error was detected.

  If an error was detected, a bit field based on the line status register of the UART is received. The following errors can be detected:
  - (0x02) Overrun Error: An overrun occured to the hardware receive FIFO. Data was lost.
  - (0x04) Parity Error: A byte was received with a parity error (Data specific bit).
  - (0x08) Framing Error: A byte was received with a framing error (Data specific bit).
  - (0x10) Break Interrupt: A break interrupt was detected (Trigger-bit).
  - (0x80) Error in receive FIFO: At least one Overrun, Parity, or Framing Error or Break Interrupt in receive FIFO.

**Debug messages** The Transmit and Receive functions offer properties to enable debug messages. The debug messages are written to the dSPACE Log and to `http://<MicroAutoBoxIII_IP_address>/action/messages`.

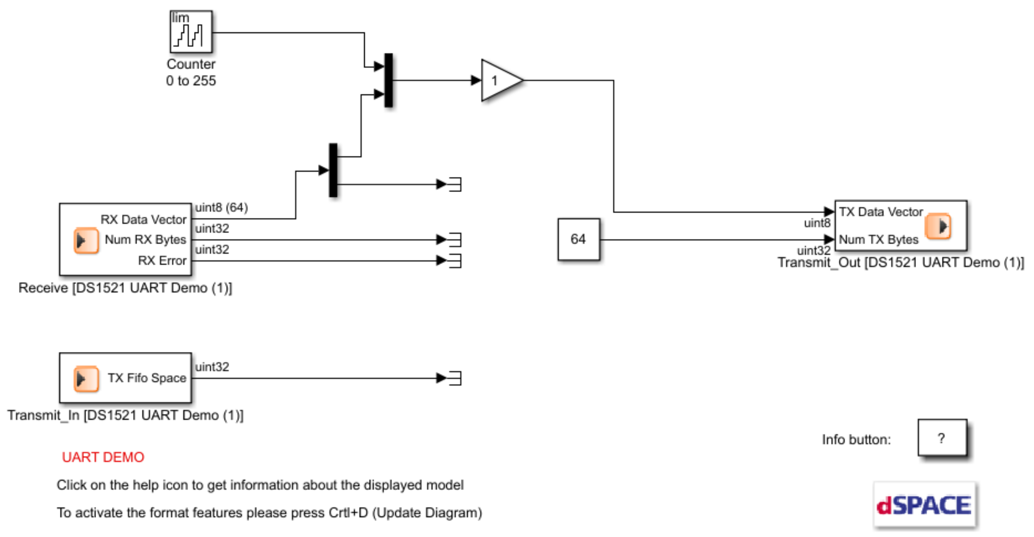
The following properties for debug messages are available:

- TX Data Info: Shows the first bytes from TX Data Vector.
- RX Data Info: Shows the first bytes from RX Data Vector.
- RX Error Info: Shows errors that were detected.

**Simulink model**

When you open the CfgUARTRS232FlowControlDemo project, the `DemoUART_DS1521.slx` Simulink model of the demo is available in the Models folder in the Project view set.

You can open the model in Simulink.


lim
Counter
0 to 255
1
RX Data Vector
Num RX Bytes
RX Error
uint8 (64)
uint32
uint32
Receive [DS1521 UART Demo (1)]
64
TX Data Vector
Num TX Bytes
uint8
uint32
Transmit_Out [DS1521 UART Demo (1)]
TX Fifo Space
uint32
Transmit_In [DS1521 UART Demo (1)]
UART DEMO
Click on the help icon to get information about the displayed model
To activate the format features please press Crtl+D (Update Diagram)
Info button:
?
dSPACE

# Building the Signal Chain for UART Communication Using an FPGA Board

**Where to go from here**

Information in this section

Example of UART Bus Communication Using an FPGA Application..........33
Port Description of the FPGA Custom Function Blocks for UART Communication..........35
Building the Demo Project and Experimenting..........40

## Example of UART Bus Communication Using an FPGA Application

**Introduction**

The demo project `CfgFPGAuartDemo` is an example of an FPGA application for implementing a configurable UART bus communication. It is not necessary to have knowledge about FPGA programming to use the example or reuse the example in a ConfigurationDesk project.

**Location of the ConfigurationDesk demo project**

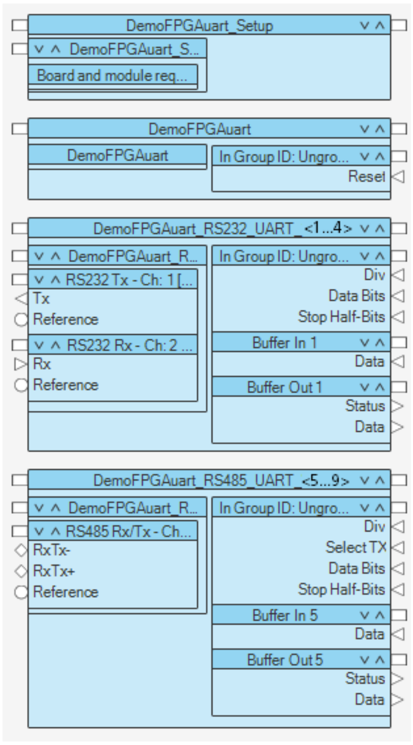
The demo project `CfgFPGAuartDemo` for implementing UART interfaces to the FPGA board is available in the `Documents` folder.

**Hardware preconditions**

The demo requires a SCALEXIO system with a DS2655 (7K160) FPGA Base Board and a DS2655M2 Digital I/O Module connected to I/O module slot 1. Refer to How to Check Hardware Resources Required for FPGA Custom Function Blocks (SCALEXIO) (ConfigurationDesk I/O Function Implementation Guide 🕮).

## Overview

The following illustration shows the FPGA custom functions to implement UART interfaces. The names of the function ports are shortened.



The demo project provides four FPGA custom functions that support RS232 communication and four FPGA custom functions that support RS485 communication.

## Reusing the demo project

You can reuse the FPGA custom function blocks of the demo in your project. The **FPGA Setup** function block and the **DemoFPGAuart** function block must be added to the signal chain. The **DemoFPGAuart_RS232_UART** function blocks and the **DemoFPGAuart_RS485_UART** function blocks are optional. For more information on the necessary assignments, refer to Building the Demo Project and Experimenting on page 40.

## Implementing UART communication in an FPGA application

The RTI FPGA Programming Blockset includes a demo with the FPGA Simulink model of this UART demo. You can use this model to implement an FPGA communication when you model an FPGA application.

Refer to Using the UART Demo Model for SCALEXIO Systems (RTI FPGA Programming Blockset Guide 📖).

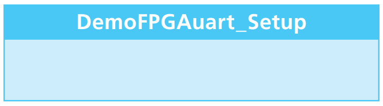
**Related topics**

Basics

Implementing FPGA Custom Function Blocks (ConfigurationDesk I/O Function Implementation Guide 🕮)

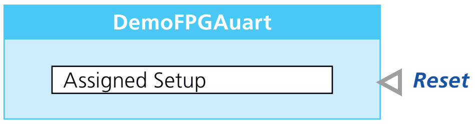
## Port Description of the FPGA Custom Function Blocks for UART Communication

**DemoFPGAuart_Setup**



This FPGA custom function block is a FPGA Setup block. The FPGA Setup block configures and initializes the access to the FPGA base board.

**DemoFPGAuart**

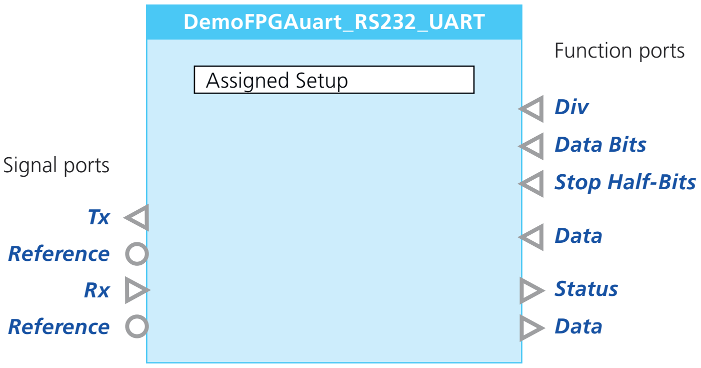


This FPGA custom function block resets all FPGA custom function blocks that provide an UART interface.

**Reset** This function inport resets the UART interfaces from within the behavior model.

| | |
|---|---|
| Value range | ▪ 1: Reset of all UART interfaces.<br>▪ 0: No reset. |

**DemoFPGAuart_RS232_UART**



This FPGA custom function block provides a RS232 interface.

**Div** This function inport provides the clock divider value from within the behavior model to set the baud rate of the interface. You can calculate the value for the clock divider to set the baud rate as follows:

$$\textit{Clock divider} = \frac{1}{\textit{Baud rate} \cdot 32\ \textit{ns}} - 1$$

For values to set common baud rates, refer to Settings for common baud rates on page 39.

| | |
|---|---|
| Value range | 125 ... $n_{max}$ |

**Data Bits** This function inport provides the number of data bits of the RS232 frames from within the behavior model.

| | |
|---|---|
| Value range | ▪ 5 ... 9<br>▪ The data bits include optional parity bits and exclude the start bit of RS232 frames. |

**Stop Half-Bits** This function inport provides the number of stop half-bits of the frames from within the behavior model.

| | |
|---|---|
| Value range | ▪ 1 ... 5<br>▪ The value specifies the number of half bits. For example: 2 specifies 1 stop bit in the frame. |

**Data (inport)** This function inport provides a vector with data values from within the behavior model. Each entry of the vector is sent with a separate frame. The control and validation of the data flow must be done within the behavior model.

**Note**

**Risk of data loss**

A new data vector from within the behavior model overwrites data values that are not sent.

- Make sure that the function block can send the received data values with the configured baud rate before new values are received from within the behavior model.

| | |
|---|---|
| Value range | ▪ 0 ... $n_{max}$ for each entry of the vector.<br>▪ The maximum value of each entry depends on the specified number of data bits.<br>▪ If parity bits are used, the entries must include the parity value.<br>▪ The vector width is 1024.<br>The function block can process data vectors with 1 ... 1024 entries. This flexibility results in a conflict that indicates the mismatching data width of the model port mapping. This conflict is normal and has no effect on the functionality and build process. |

**Status** This function outport provides a vector with status information on the IOCNET bus that sends received data from the FPGA application to the behavior model.

| | |
|---|---|
| Value range | ▪ The vector width is 3.<br>▪ First entry (Status[0]): Provides the number of valid elements to be sent to the behavior model.<br>▪ Second entry (Status[1]): Indicates whether the current buffer contains new or old values. It is 1 if the buffer contains new values.<br>▪ Third entry (Status[2]): Indicates whether a buffer overflow occurred. At least one buffer was not read and its data was lost before the currently read buffer was filled. |

**Data (outport)** This function outport provides a vector with the received data values to be sent to the behavior model.

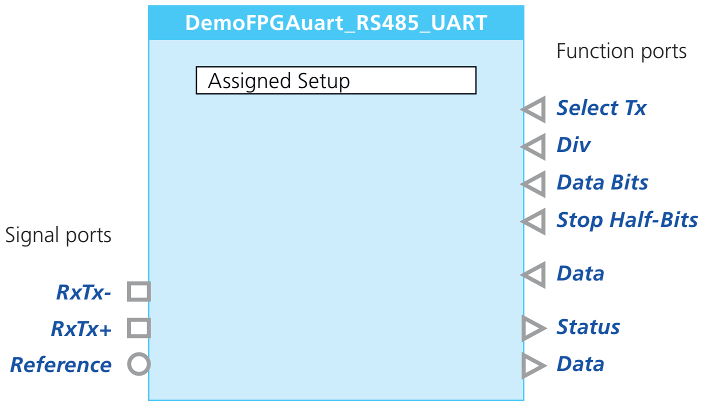
| | |
|---|---|
| Value range | ▪ 0 ... $n_{max}$ for each entry of the vector.<br>▪ The maximum value of each entry depends on the specified number of data bits.<br>▪ If parity bits are used, the entries include the parity value.<br>▪ The vector width is 1024.<br>The function block can process data vectors with 1 ... 1024 entries. This flexibility results in a conflict that indicates the mismatching data width of the model port mapping. This conflict is normal and has no effect on the functionality and build process. |

**Tx** This signal port represents the electrical connection point of the logical signal for the transmit channel.

**Rx** This signal port represents the electrical connection point of the logical signal for the receive channel.

**Reference** This signal port is a reference port and provides the reference signal for the Tx or Rx signal ports.

## DemoFPGAuart_RS485_UART



This FPGA custom function block provides an RS485 interface.

**Select Tx** This function inport enables the electrical interface from within the behavior model to output data values to the RS485 network.

| | |
|---|---|
| Value range | ▪ 0: The output is set to the high-impedance state (tri-state). The function block can receive data from the RS485 network.<br>▪ 1: The output is enabled and the function block can send data to the RS485 network. |

**Div** This function inport provides the clock divider value from within the behavior model to set the baud rate of the interface. You can calculate the value for the clock divider to set the baud rate as follows:

$$\textit{Clock divider} = \frac{1}{\textit{Baud rate} \cdot 32\ \textit{ns}} - 1$$

For values to set common baud rates, refer to Settings for common baud rates on page 39.

| | |
|---|---|
| Value range | 1 ... $n_{max}$ |

**Data Bits** This function inport provides the number of data bits of the RS485 frames from within the behavior model.

| | |
|---|---|
| Value range | ▪ 5 ... 9<br>▪ The data bits include optional parity bits and exclude the start bit of RS485 frames. |

**Stop Half-Bits** This function inport provides the number of stop bits in the frames from within the behavior model.

| | |
|---|---|
| Value range | ▪ 1 ... 5<br>▪ The value specifies the number of half bits. For example: 2 specifies 1 stop bit in the frame. |

**Data (inport)** This function inport provides a vector with data values from within the behavior model. Each entry of the vector is sent with a separate frame. The control and validation of the data flow must be done within the behavior model.

**Note**

**Risk of data loss**

A new data vector from within the behavior model overwrites data values that are not sent.

- Make sure that the function block can send the received data values with the configured baud rate before new values are received from within the behavior model.

| | |
|---|---|
| Value range | ▪ 0 ... $n_{max}$ for each entry of the vector.<br>▪ The maximum value of each entry depends on the specified number of data bits.<br>▪ If parity bits are used, the entries must include the parity value.<br>▪ The vector width is 1024.<br>The function block can process data vectors with 1 ... 1024 entries. This flexibility results in a conflict that indicates the |

| | |
|---|---|
| | mismatching data width of the model port mapping. This conflict is normal and has no effect on the functionality and build process. |

**Status** This function outport provides a vector with status information on the IOCNET bus that sends received data from the FPGA application to the behavior model.

| | |
|---|---|
| Value range | ▪ The vector width is 3.<br>▪ First entry (Status[0]): Provides the number of valid elements to be sent to the behavior model.<br>▪ Second entry (Status[1]): Indicates whether the current buffer contains new or old values. It is 1 if the buffer contains new values.<br>▪ Third entry (Status[2]): Indicates whether a buffer overflow occurred. At least one buffer was not read and its data was lost before the currently read buffer was filled. |

**Data (outport)** This function outport provides a vector with the received data values to be sent to the behavior model.

| | |
|---|---|
| Value range | ▪ 0 ... $n_{max}$ for each entry of the vector.<br>▪ The maximum value of each entry depends on the specified number of data bits.<br>▪ If parity bits are used, the entries include the parity value.<br>▪ The vector width is 1024.<br>The function block can process data vectors with 1 ... 1024 entries. This flexibility results in a conflict that indicates the mismatching data width of the model port mapping. This conflict is normal and has no effect on the functionality and build process. |

**RxTx-** This signal port is a bidirectional port. It represents the inverted signal of the electrical connection point for the logical signal.

**RxTx+** This signal port is a bidirectional port. It represents the non-inverted signal of the electrical connection point for the logical signal.

**Reference** This signal port is a reference port and provides the reference for the RxTx- and RxTx+ signal ports.

**Settings for common baud rates**

The following table gives you clock divider values for the to set the UART interface to a common baud rate:

| Desired Baud Rate | Clock Divider Value |
|---|---|
| 9600 | 3254 |
| 19200 | 1627 |
| 38400 | 813 |
| 57600 | 542 |
| 115200 | 270 |
| 1000000 | 30 |

**Related topics**

Basics

Implementing FPGA Custom Function Blocks (ConfigurationDesk I/O Function Implementation Guide 📖)

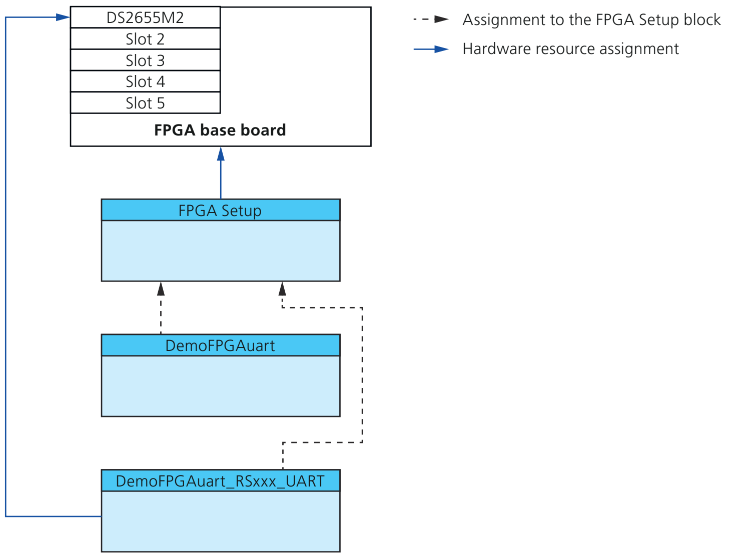
## Building the Demo Project and Experimenting

**Introduction**

To experiment with the demo project, you have to assign the FPGA application to your hardware before you can build and download the application for experimenting. To experiment with the demo application, a ControlDesk experiment is available.

**Assignments of the blocks**

The **FPGA Setup** block and the FPGA custom function block that provide the UART interface (**DemoFPGAuart_RS232_UART** and **DemoFPGAuart_RS485_UART** function blocks) must be assigned to your FPGA hardware resources as shown in the following illustration. For more information, refer to Assigning Hardware Resources to Function Blocks (ConfigurationDesk Real-Time Implementation Guide 📖) and Working with Hardware Topologies (ConfigurationDesk Real-Time Implementation Guide 📖).



Furthermore, all FPGA custom function blocks must be assigned to the **FPGA Setup** block via the **Assigned Setup** property. This is already done in the demo project.

For more information on the assignment to the **FPGA Setup** block, refer to Types of FPGA Applications (ConfigurationDesk I/O Function Implementation Guide 🕮).

**ControlDesk experiment**

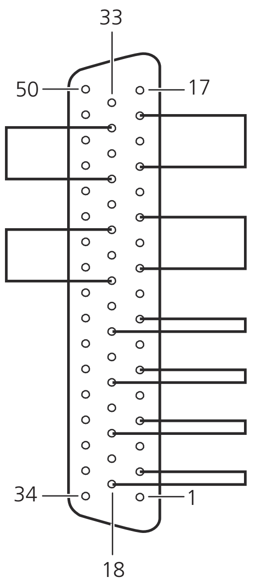
A ControlDesk project with an experiment using the UART function block is available in the `Documents` folder:

`.\Documents\dSPACE\ConfigurationDesk\<version>\FPGAuartDemo\CDNG_FPGAuartDemo\`

You can open the project in ControlDesk using the **Open Project + Experiment** command. For details, refer to How to Open a Project and Experiment (ControlDesk Project and Experiment Management 🕮).

In ControlDesk you can start the real-time application and observe the UART communication.

**Recommended cable harness for experimenting** If you use the demo project and experiment, it is useful to connect the UART interfaces of two function blocks. This lets you send data with one function block and receive the data with the other. The following illustration shows you the pins to be connected:



**Related topics**

Basics

ControlDesk Introduction and Overview

# ConfigurationDesk UART API Reference

**Introduction**

Reference information on API functions which you can use to program a UART interface in a SCALEXIO or MicroAutoBox III system. The programming language of the API is C++.

**Where to go from here**

Information in this section

DsCIoFuncUart Class....................44

Information in other sections

Serial Interface Connection (SCALEXIO – Hardware and Software Overview 🕮)
To connect external devices to the SCALEXIO system via a serial interface, you must implement the corresponding communication in the real-time model.

## DsCIoFuncUart Class

**Introduction**

The DsCIoFuncUart class implements an I/O driver for the UART functionality.

**Where to go from here**

Information in this section

DsNIoFuncUart Namespace....................44

DsCIoFuncUart Methods....................60

## DsNIoFuncUart Namespace

**Introduction**

The `DsNIoFuncUart` module provides the namespaces for the `DsCIoFuncUart` class.

**Where to go from here**

Information in this section

General Information on the DsNIoFuncUart Namespace....................45
Providing information on the namespaces of constants used in the methods of the DsCIoFuncUart class.

DsNIoFuncUart::Baudrate....................46
Default value and absolute limits for the baud rate.

DsNIoFuncUart::ChannelStatus....................46
Channel status bits which indicates the current state of the driver.

DsNIoFuncUart::ChannelType....................47
Type of channel.

DsNIoFuncUart::Databits....................48
Constants which specify the number of data bits used in a data frame.

DsNIoFuncUart::ErrorCode....................48
Error codes for the DsCIoFuncUart module.

DsNIoFuncUart::ErrorStatus....................50
Constants for UART error bits.

DsNIoFuncUart::Event....................50
Constants for UART events that can trigger application tasks.

DsNIoFuncUart::FlowControlMode....................52
Constants and default values for flow control mode.

DsNIoFuncUart::Parity.....53
Constants for specifying the parity method used in a data frame.

DsNIoFuncUart::RxFifoBlockSize.....53
Default value and limit for the receive FIFO block size.

DsNIoFuncUart::RxFifoMode.....54
Constants to specify the receive FIFO overwrite mode.

DsNIoFuncUart::RxFifoSize.....54
Constants for the default value and lower limit for the size of a receive FIFO.

DsNIoFuncUart::RxFifoTriggerLevel.....55
Constants for specifying the default value and the lower limit for the trigger level of a receive FIFO.

DsNIoFuncUart::Stopbits.....56
Constants for specifying the number of stop bits used in a data frame.

DsNIoFuncUart::Termination.....56
Constants and default values for setting the termination types.

DsNIoFuncUart::Transceiver.....57
Constants and default values for selecting the transceiver types.

DsNIoFuncUart::UartLineStatus.....58
Constants for the line status bits of the UART.

DsNIoFuncUart::UartModemStatus.....59
Constants for the modem status bits of the UART.

DsNStdErrorCode.....60
Standard error codes used by the DsCIoFuncUart module.

Information in other sections

DsCIoFuncUart Methods.....60

## General Information on the DsNIoFuncUart Namespace

**Namespace**

Many of the namespaces define constants, that you can use to specify parameter values for the different set methods. For these namespaces, also the following standard constants may be defined:

- `Default`: is the default value that is used if the set method for the corresponding parameter is not called.
- `UpperLimit`: is the maximum value that can be set with the corresponding set method.

- `LowerLimit`: is the minimum value that can be set with the corresponding set method.

Other namespaces define constants that can be used to check return values of get methods, for example.

## DsNIoFuncUart::Baudrate

**Namespace**

`DsNIoFuncUart::Baudrate`

**Description**

The namespace contains a default value and absolute limits for the baud rate. If special transceivers are used, the baud rate may be further limited.

**Variables**

| Variable | Description |
|---|---|
| `const UInt32 LowerLimit = 5` | Minimum baud rate. |
| `const UInt32 UpperLimit = 1000000` | Maximum baud rate. |
| `const UInt32 Default = 115200` | Default value: 115200 baud. |

**Related topics**

References

DsCIoFuncUart::applySettings........ 62
DsCIoFuncUart::setBaudrate........ 79

## DsNIoFuncUart::ChannelStatus

**Namespace**

`DsNIoFuncUart::ChannelStatus`

**Description**

The namespace contains constants for channel status bits which indicates the current state of the driver. You can use the constants to check the return value of the `DsCIoFuncUart::getChannelStatus` method to retrieve the current state of the driver.

## Variables

| Variable | Description |
|---|---|
| `const UInt32 Running = 0x00000001UL` | Running: `start` method was called, the driver is running. |
| `const UInt32 BaudrateSetupPending = 0x00000002UL` | The baud rate was changed and is not stable yet. |
| `const UInt32 ChannelSetupPending = 0x00000004UL` | Transceiver and/or termination were changed. The switching sequence is not finished yet. |

## Related topics

References

DsCIoFuncUart::getChannelStatus........................................................................................... 67

## DsNIoFuncUart::ChannelType

**Namespace**

`DsNIoFuncUart::ChannelType`

**Description**

The namespace contains the type of a channel. It is used if two channels are bound to implement hardware flow control. If you want to use hardware flow control, two consecutive channels of a bus board must be bound to provide the required number of electrical signals. For details, refer to DsCIoFuncUart::bindIoChannel on page 63 and Creating and Adjusting the Source Code Files (Serial Communication) (ConfigurationDesk Custom I/O Function Implementation Guide 📖).

**Variables**

| Variable | Description |
|---|---|
| `const UInt32 Data = 0` | Channel is used for data lines (RX/TX). |
| `const UInt32 Handshake = 1` | Channel is used for hardware handshake lines (RTS/CTS). |

## DsNIoFuncUart::Databits

**Namespace**

`DsNIoFuncUart::Databits`

**Description**

The namespace contains constants which specify the number of data bits which are used in a data frame. You can use these constants with the `DsCIoFuncUart::setDatabits` method to specify the desired number of data bits.

**Variables**

| Variable | Description |
|---|---|
| `const UInt32 LowerLimit = 5` | At least 5 data bits have to be used. |
| `const UInt32 UpperLimit = 8` | No more than 8 data bits can be used. |
| `const UInt32 Default = UpperLimit` | Default value: 8 data bits. |

**Related topics**

References

DsCIoFuncUart::setDatabits........................................................................................................81

## DsNIoFuncUart::ErrorCode

**Namespace**

`DsNIoFuncUart::ErrorCode`

**Description**

The namespace contains error codes for the DsCIoFuncUart module.

For standard error codes used by the DsCIoFuncUart module, refer to DsNStdErrorCode on page 60.

**Variables**

All error codes are added to a base Id:

```
const UInt32 ClassId = DsNClassId::CommonScope::DsIoFuncUart::DsCIoFuncUart
```

| Variable | Description |
|---|---|
| `const UInt32 TxFifoSizeRange = ClassId + 0` | The `TxFifoSize` parameter is out of range. |
| `const UInt32 RxFifoSizeRange = ClassId + 1` | The `RxFifoSize` parameter is less than the lower limit. |
| `const UInt32 RxFifoModeRange = ClassId + 2` | The `RxFifoMode` parameter is out of range. |
| `const UInt32 RxFifoBlockSizeMinRange = ClassId + 3` | The `RxFifoBlockSize` parameter is less than the lower limit. |

| Variable | Description |
|---|---|
| `const UInt32 RxFifoBlockSizeMaxRange = ClassId + 4` | The `RxFifoBlockSize` parameter is greater than (RxFifoSize-1). |
| `const UInt32 UartBaudrateRange = ClassId + 5` | The `Baudrate` parameter is out of range. |
| `const UInt32 DatabitsRange = ClassId + 6` | The `Databits` parameter is out of range. |
| `const UInt32 StopbitsRange = ClassId + 7` | The `Stopbits` parameter is out of range. |
| `const UInt32 ParityRange = ClassId + 8` | The `Parity` parameter is out of range. |
| `const UInt32 RxFifoTriggerLevelMinRange = ClassId + 9` | The `RxFifoTriggerLevel` parameter is less than the lower limit. |
| `const UInt32 RxFifoTriggerLevelMaxRange = ClassId + 10` | The `RxFifoTriggerLevel` parameter is greater than (RxFifoSize-1). |
| `const UInt32 FlowControlModeRange = ClassId + 11` | The `FlowControlMode` parameter is out of range. |
| `const UInt32 TransceiverRange = ClassId + 12` | The `Transceiver` parameter is out of range. |
| `const UInt32 TerminationRange = ClassId + 13` | The `Termination` parameter is out of range. |
| `const UInt32 NoDataChannelBound = ClassId + 14` | No (valid) I/O channel is bound to the I/O driver. |
| `const UInt32 ChannelRange = ClassId + 15` | The channel number is out of range. |
| `const UInt32 ChannelUsed = ClassId + 16` | The channel is already in use. |
| `const UInt32 NoHandshakeChannelBound = ClassId + 17` | No (valid) handshake channel bound to the driver. Hardware flow control is not possible. |
| `const UInt32 InvalidHandshakeChannel = ClassId + 18` | Invalid handshake channel bound to the driver with data channel. |
| `const UInt32 EventIdMaxRange = ClassId + 19` | The `EventId` parameter is greater than the upper limit. |
| `const UInt32 TestFunctionCalled = ClassId + 20` | A prohibited test function was called. |
| `const UInt32 TransceiverBaudrateRange = ClassId + 21` | The `Baudrate` parameter is out of the transceiver specific range. |
| `const UInt32 TerminationMismatch = ClassId + 22` | The selected termination is not supported for the selected transceiver. |
| `const UInt32 NoTargetNode = ClassId + 23` | Configuration error. The target node could not be found. |
| `const UInt32 CommunicationTimeout = ClassId + 24` | Communication error. RTRPC connection timeout. |
| `const UInt32 HwTimeout = ClassId + 25` | Initialization error. A hardware timeout occurred. |
| `const UInt32 NoChConfig = ClassId + 26` | Initialization error. No channel configuration found. |
| `const UInt32 NoIocnetNode = ClassId + 27` | Initialization error. No IOCNET node specified. |
| `const UInt32 NoInventory = ClassId + 28` | Initialization error. No inventory data specified. |
| `const UInt32 MappingFailure = ClassId + 29` | Initialization error. Failed to map UART memory. |
| `const UInt32 InitIntFailure = ClassId + 30` | Initialization error. Failed to initialize UART interrupt. |
| `const UInt32 FeatureRange = ClassId + 31` | Feature request error. The requested feature is unknown. |
| `const UInt32 ParameterRange = ClassId + 32` | Parameter request error. The requested parameter is unknown. |
| `const UInt32 InventoryNotRead = ClassId + 33` | The inventory was not read before. |
| `const UInt32 RxFifoTriggerLevelInvalid = ClassId + 34` | Configuration error. The specified RxFifoTriggerLevel is not supported. |
| `const UInt32 UnsupportedTransceiver = ClassId + 35` | The selected transceiver is not supported on this platform. |
| `const UInt32 UnsupportedTermination = ClassId + 36` | The selected termination is not supported on this platform. |

| Variable | Description |
|---|---|
| `const UInt32 RxFifoSizeInvalidRange = ClassId + 37` | Configuration error. Specified parameter RxFifoSize is out of range. |
| `const UInt32 InvalidChannelType = ClassId + 38` | Initialization error. Selected channel type is not supported. |
| `const UInt32 IpControlInitFailure = ClassId + 39` | Initialization error. Failed to initialize IP-Control communication. |
| `const UInt32 NullPointer = ClassId + 40` | The specified argument is a NULL pointer. |
| `const UInt32 RxFifoBlockSizeRange = ClassId + 41` | Configuration error. Selected value for parameter RxFifoBlockSize is out of range. |
| `const UInt32 RxFifoTriggerLevelRange = ClassId + 42` | Configuration error. Selected value for parameter RxFifoTriggerLevel is out of range. |
| `const UInt32 InventoryReadError = ClassId + 43` | Error reading the inventory. |

## DsNIoFuncUart::ErrorStatus

**Namespace**

`DsNIoFuncUart::ErrorStatus`

**Description**

The namespace contains constants for UART error bits. These bits indicate an error in communication, not an error in the driver. See also DsCIoFuncUart::getErrorStatus on page 68.

**Variables**

| Variable | Description |
|---|---|
| `const UInt32 NoError = 0x00000000UL` | Error status: No error occurred. |
| `const UInt32 RxFifoOverflow = 0x00000001UL` | Error status: Receive FIFO overflow (hardware or software). Data is lost. |

## DsNIoFuncUart::Event

**Namespace**

`DsNIoFuncUart::Event`

**Description**

The namespace contains constants for UART events that can trigger application tasks. You must use the constants to specify the source within the UART hardware that shall trigger event-triggered application tasks. For details, refer to Defining I/O Events (ConfigurationDesk Custom I/O Function Implementation Guide 📖).

## Variables

| Variable | Description |
|---|---|
| `const UInt32 LineStatusChanged = 0x00000000U` | The line status has changed, see also DsNIoFuncUart::UartLineStatus on page 58. |
| `const UInt32 ModemStatusChanged = 0x00000001U` | The modem status has changed, see also DsNIoFuncUart::UartModemStatus on page 59. |
| `const UInt32 RxFifoLevelReached = 0x00000002U` | The selected receive FIFO level was reached. |
| `const UInt32 TxFifoEmpty = 0x00000003U` | The transmit FIFO empty. |

## Example

The UART RS232 FlowControl Custom Function example shows how the constants are used.

```
/** <!------------------------------------------------------------------------->
* Macro binds an UART driver object to an event as a trigger (DO NOT CHANGE!)
*
* @parameters
*  @param P_DRIVER Pointer to the generated UART driver object (will be declared)
*  @param EVENT Name of the function event declared in UART_RS232_FlowControl.xml
*
* @note
*  To use in create function
*
*<!------------------------------------------------------------------------->*/
#define UART_DRIVER_TRIGGER_EVENT(P_DRIVER,EVENT)                        \
  pBlockInstance->EVENT##_driver = P_DRIVER;

// ...

/** <!------------------------------------------------------------------------->
* CustomIoFunction Create access point
*
* @description
*  This function is called for every created instance of the
*  UART_RS232_FlowControl CustomIoFunction. See the remarks below
*  for the detailed description.
*
* @parameters
*  @param ErrorList ErrorList to report errors
*  @param pBlockInstance Pointer to the BlockInstance buffer that is used
*       to handle different instances of the CustomIoFunction
*
*<!------------------------------------------------------------------------->*/
void ucf_Uart_Create(DsTErrorList ErrorList, ucf_UART_RS232_FlowControlStruct_T* pBlockInstance)
{

   // ...

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

   // ...

}
```

For another example, refer to DsCIoFuncUart::setTriggerEnable on page 87.

## DsNIoFuncUart::FlowControlMode

| | |
|---|---|
| **Namespace** | `DsNIoFuncUart::FlowControlMode` |
| **Description** | The namespace contains constants and default values for flow control mode. Refer to DsCIoFuncUart::setFlowControlMode on page 82. |

**Variables**

| Variable | Description |
|---|---|
| `const UInt32 None = 0` | No flow control is selected. |
| `const UInt32 Hardware = 1` | Hardware flow control is selected. |
| `const UInt32 Default = None` | Default: No flow control is selected. |

## DsNIoFuncUart::Parity

**Namespace**

`DsNIoFuncUart::Parity`

**Description**

The namespace contains constants for specifying the parity method used in a data frame. For details, refer to DsCIoFuncUart::setParity on page 84.

**Variables**

| Variable | Description |
|---|---|
| `const UInt32 No = 0` | No parity bit is added to the data frame. |
| `const UInt32 Odd = 1` | An odd parity bit is added to the data frame. |
| `const UInt32 Even = 2` | An even parity bit is added to the data frame. |
| `const UInt32 Default = No` | Default value: No parity. |

## DsNIoFuncUart::RxFifoBlockSize

**Namespace**

`DsNIoFuncUart::RxFifoBlockSize`

**Description**

The namespace contains the default value and limit for the receive FIFO block size. This block size is used in case of an FIFO overrun. For details, refer to DsCIoFuncUart::setRxFifoBlockSize on page 88.

**Variables**

| Variable | Description |
|---|---|
| `const UInt32 LowerLimit = 1` | Minimum block size of receive FIFO. |
| `const UInt32 UpperLimit = 4095` | Maximum block size of receive FIFO. |
| `const UInt32 Default = 1` | Default block size of receive FIFO. |

## DsNIoFuncUart::RxFifoMode

**Namespace**

`DsNIoFuncUart::RxFifoMode`

**Description**

The namespace contains constants to specify the receive FIFO overwrite mode. You can use the constants with the `DsCIoFuncUart::setRxFifoMode` method to specify the behavior of the receive FIFO if it overruns.

**Variables**

| Variable | Description |
|---|---|
| `const UInt32 DiscardNew = 0` | New data is discarded if the FIFO overruns. |
| `const UInt32 OverwriteOldest = 1` | Oldest data is overwritten if the FIFO overruns. |
| `const UInt32 Default = DiscardNew` | Default value. |

**Related topics**

References

DsCIoFuncUart::setRxFifoMode........................................................................................ 90

## DsNIoFuncUart::RxFifoSize

**Namespace**

`DsNIoFuncUart::RxFifoSize`

**Description**

The namespace contains the default value and the lower limit for the size of a receive FIFO. You can set the size of the receive FIFO using the `DsCIoFuncUart::setRxFifoSize` method.

**Variables**

| Variable | Description |
|---|---|
| `const UInt32 LowerLimit = 256` | Minimum size of a receive FIFO. |
| `const UInt32 UpperLimit = 4096` | Maximum size of a receive FIFO. |
| `const UInt32 Default = 1024` | Default size of a receive FIFO. |

**Related topics**

References

DsCIoFuncUart::setRxFifoSize........................................................................................................ 91

## DsNIoFuncUart::RxFifoTriggerLevel

**Namespace**

`DsNIoFuncUart::RxFifoTriggerLevel`

**Description**

The namespace contains the default value and the lower limit for the trigger level of a receive FIFO. You can set the trigger level of a receive FIFO using the `DsCIoFuncUart::setRxFifoTriggerLevel` method within the range 1 .. (receive FIFO size - 1).

The upper limit depends on the used channel type:

| Channel Type | Limit (FIFO size -1) |
|---|---|
| Bus 1 | 1023 |
| UART 1 | 511 |
| UART 4 | 511 |
| UART 5[1)] | 14 |
| UART 6 | 1023 |

[1)] For this channel type, only four constant values are available. Refer to Variables on page 55.

**Variables**

| Variable | Description |
|---|---|
| `const UInt32 LowerLimit = 1` | The minimum trigger level of a receive FIFO. |
| `const UInt32 Default = 1` | The default trigger level of a receive FIFO. |

For the channel type UART 5 (SCALEXIO Processing Unit), only the following values are available:

| Variable |
|---|
| `const UInt32 Bytes1 = 1` |
| `const UInt32 Bytes4 = 4` |
| `const UInt32 Bytes8 = 8` |
| `const UInt32 Bytes14 = 14` |

**Related topics**

References

DsCIoFuncUart::setRxFifoTriggerLevel........................................ 92

## DsNIoFuncUart::Stopbits

**Namespace**

`DsNIoFuncUart::Stopbits`

**Description**

The namespace contains the constants for specifying the number of stop bits used in a data frame. You can set the number of stop bit using the `DsCIoFuncUart::setStopbits` method.

**Variables**

| Variable | Description |
|---|---|
| `const UInt32 One = 0` | Use one stop bit. |
| `const UInt32 Two = 1` | Use two stop bits. |
| `const UInt32 Default = One` | Default value: One stop bit. |

**Related topics**

References

DsCIoFuncUart::setStopbits........................................ 83

## DsNIoFuncUart::Termination

**Namespace**

`DsNIoFuncUart::Termination`

**Description**

The namespace contains constants and default values for setting the termination types. You can select the desired termination using the `DsCIoFuncUart::setTermination` method.

**Variables**

| Variable | Description |
|---|---|
| `const UInt32 None = 0` | No termination selected. |
| `const UInt32 Parallel_AB = 1` | 90 Ohms parallel to bus lines A and B |
| `const UInt32 Parallel_CD = 2` | 90 Ohms parallel to bus lines C and D |
| `const UInt32 FtCAN_A = 3` | Ft-CAN 560 Ohm |
| `const UInt32 FtCAN_B = 4` | Ft-CAN 5.6 kOhm |
| `const UInt32 LIN_Slave = 5` | LIN slave termination |
| `const UInt32 LIN_Master = 6` | LIN master termination |
| `const UInt32 Parallel_ABCD = 7` | 90 Ohms parallel to bus lines A and B and to bus lines C and D |
| `const UInt32 UART = 8` | 120 Ohms termination for the UART transceivers |
| `const UInt32 KLine = 9` | K-Line termination |
| `const UInt32 Default = None` | Default: No termination selected. |

**Related topics**

References

DsCIoFuncUart::setTermination............................................................................................ 85

## DsNIoFuncUart::Transceiver

**Namespace**

`DsNIoFuncUart::Transceiver`

**Description**

The namespace contains constants and default values for selecting the transceiver types. You can use the `DsCIoFuncUart::setTransceiver` method to select the desired transceiver.

**Variables**

| Variable | Description |
|---|---|
| `const UInt32 None = 0` | No transceiver selected |
| `const UInt32 CAN_ISO11898_2 = 1` | High-speed CAN transceiver |
| `const UInt32 CAN_ISO11898_3 = 2` | Fault-tolerant CAN transceiver |
| `const UInt32 LIN = 3` | LIN / K-Line transceiver |
| `const UInt32 RS232 = 4` | RS232 transceiver |
| `const UInt32 RS422 = 5` | RS422 transceiver |
| `const UInt32 RS485 = 6` | RS485 transceiver |
| `const UInt32 FlexRay = 7` | FlexRay transceiver |

| Variable | Description |
|---|---|
| `const UInt32 TTL = 8` | TTL transceiver |
| `const UInt32 PiggyBack = 9` | PiggyBack transceiver |
| `const UInt32 Default = None` | Default: No transceiver selected |

**Related topics**

References

DsCIoFuncUart::setTransceiver.......................................................................................... 86

## DsNIoFuncUart::UartLineStatus

**Namespace**

`DsNIoFuncUart::UartLineStatus`

**Description**

The namespace contains constants for the line status bits of the UART. They represent the LSR bits of a standard 16550 UART.

The bits shows the status of different parts of a UART. The reset behavior of the individual bits are different:

- Data-specific bits: These bits show the status of the first byte in the receive FIFO.
- FIFO-specific bits: These bits show the status of the entire FIFO.
- UART-specific bits: These bits show the status of the UART hardware.
- Trigger bits: These bits are set if the specific condition occurs once. They are reset if the status is read with the `DsCIoFuncUart::getLineStatus` method.

**Variables**

| Variable | Description |
|---|---|
| `const UInt8 DataReady = 0x01U` | Data ready: Data is available in the receive FIFO (FIFO-specific bit). |
| `const UInt8 OverrunError = 0x02U` | Overrun error: An overrun occurred in the hardware receive FIFO. Data was lost (trigger bit). |
| `const UInt8 ParityError = 0x04U` | Line status bit: Parity error: A byte was received with a parity error (data-specific bit). |
| `const UInt8 FramingError = 0x08U` | Framing error: A byte was received with a framing error (data-specific bit). |
| `const UInt8 BreakInterrupt = 0x10U` | Break interrupt: A break interrupt was detected (trigger bit). |
| `const UInt8 TxEmpty = 0x20U` | Transmitter holding register empty (UART-specific bit). |
| `const UInt8 TxFifoEmpty = 0x40U` | Transmit FIFO empty (but maybe one byte left in the Transmitter Holding Register) (UART-specific bit). |

| Variable | Description |
|---|---|
| `const UInt8 RxFifoError = 0x80U` | Error in receive FIFO: At least one overrun, parity, framing error or break interrupt in receive FIFO (FIFO-specific bit). |

**Related topics**

References

DsCIoFuncUart::getLineStatus........................................ 69

## DsNIoFuncUart::UartModemStatus

**Namespace**

`DsNIoFuncUart::UartModemStatus`

**Description**

The namespace contains constants for the modem status bits of the UART. They represent the MSR bits of a standard 16550 UART. You can read the modem status using the **`DsCIoFuncUart::getModemStatus`** method.

**Note**

All modem status bits of a 16550 UART are realized, but only the CTS signal is externally connected. So only the DeltaCTS- and the CTS status bits provide valid information.

**Variables**

| Variable | Description |
|---|---|
| `const UInt8 DeltaCTS = 0x01U` | Change at the CTS (clear to send) input. |
| `const UInt8 DeltaDSR = 0x02U` | Change at the DSR (data set ready) input. |
| `const UInt8 TrailingEdgeRI = 0x04U` | Change at the edge RI (ring indicator). |
| `const UInt8 DeltaDCD = 0x08U` | Change at the DTD (data carrier detect) input. |
| `const UInt8 CTS = 0x10U` | CTS (clear to send) input state |
| `const UInt8 DSR = 0x20U` | DSR (data set ready) input state |
| `const UInt8 RI = 0x40U` | RI (ring indicator) input state |
| `const UInt8 DCD = 0x80U` | DCD (data carrier detect) input state |
| `const UInt8 Default = 0` | Default: No bit set |

**Related topics**

References

DsCIoFuncUart::getModemStatus........ 71

## DsNStdErrorCode

**Namespace**

`DsNStdErrorCode`

**Description**

The namespace specifies standard error codes which are used by the `DsCIoFuncUart` module.

For error codes specified for the `DsCIoFuncUart` module, refer to DsNIoFuncUart::ErrorCode on page 48.

**Variables**

All error codes are added to a base Id: `StandardErrorId`

| Variable | Description |
|---|---|
| `const UInt32 MemoryAllocation = StandardErrorId + 2` | Memory allocation failed. It is probably not enough memory available for the requested operation. |

## DsCIoFuncUart Methods

**Introduction**

The `DsCIoFuncUart` class realizes an I/O driver for the UART functionality of the SCALEXIO system.

**Where to go from here**

Information in this section

DsCIoFuncUart::applySettings........ 62
To apply preassigned settings to the driver and the hardware.

DsCIoFuncUart::bindIoChannel........ 63
To bind a single I/O channel to the driver.

DsCIoFuncUart::create........ 66
To instantiate a DsCIoFuncUart driver object.

DsCIoFuncUart::getChannelStatus....67
To get the status of the channel.

DsCIoFuncUart::getErrorStatus....68
To get the error status of UART communication.

DsCIoFuncUart::getLineStatus....69
To get the line status.

DsCIoFuncUart::getModemStatus....71
To get the modem status.

DsCIoFuncUart::getRxFifoLevel....72
To get the number of bytes in the software receive FIFO.

DsCIoFuncUart::getRxFifoSize....72
To get the actual size of the software receive FIFO.

DsCIoFuncUart::getTxFifoSpace....73
To get the space left in the transmit FIFO.

DsCIoFuncUart::receive....74
To check for received user data.

DsCIoFuncUart::receiveTerm....76
To get received user data until a termination character is detected.

DsCIoFuncUart::resetRxFifo....77
To clear the receive FIFO.

DsCIoFuncUart::send....78
To send data.

DsCIoFuncUart::setBaudrate....79
To set the baud rate for the channel.

Baud Rates for Onboard UART of SCALEXIO Processing Hardware....80

DsCIoFuncUart::setDatabits....81
To set the number of data bits.

DsCIoFuncUart::setFlowControlMode....82
To set the flow control mode.

DsCIoFuncUart::setStopbits....83
To set the number of stop bits in a data frame.

DsCIoFuncUart::setParity....84
To set the parity for the data frames.

DsCIoFuncUart::setTermination....85
To select a termination for the channel.

DsCIoFuncUart::setTransceiver....86
To select a transceiver for a channel.

DsCIoFuncUart::setTriggerEnable....87
To enable or disable individual trigger sources

DsCIoFuncUart::setRxFifoBlockSize........................................................88
To set the block size for a receive FIFO in overwrite mode.

DsCIoFuncUart::setRxFifoMode............................................................90
To set the mode of the receive FIFO if an overrun occurs.

DsCIoFuncUart::setRxFifoSize..............................................................91
To set the size of the receive FIFO.

DsCIoFuncUart::setRxFifoTriggerLevel....................................................92
To set the trigger level for the software receive FIFO.

DsCIoFuncUart::start.............................................................................93
To start UART operation.

DsCIoFuncUart::stop..............................................................................94
To stop UART operation.

Information in other sections

DsNIoFuncUart Namespace.....................................................................44

## DsCIoFuncUart::applySettings

**Syntax**

```
DsTError applySettings   (
   DsTErrorList   ErrorList)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To apply preassigned settings to the driver and the hardware.

**Description**

This method must be used to apply the settings previously made for the driver and the corresponding hardware. When the settings are applied, the following constraints are checked:

- RxFifoBlockSize < RxFifoSize
- RxFifoTriggerLevel < RxFifoSize
- Hardware flow control is only possible if a handshake channel was bound to the driver.
- Transceiver-specific baud rate limits
- Termination constraints: Not all combinations of transceivers and terminations are allowed

The specified receive FIFO size can be increased automatically for internal reasons. For details, refer to DsCIoFuncUart::setRxFifoSize on page 91 and DsCIoFuncUart::setRxFifoBlockSize on page 88.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**Return value**

- 0 if no error occurred.
- Otherwise error code as specified in `DsNIoFuncUart::ErrorCode`:
  - `DsNIoFuncUart::ErrorCode::RxFifoBlockSizeMaxRange`
  - `DsNIoFuncUart::ErrorCode::RxFifoTriggerLevelMaxRange`
  - `DsNIoFuncUart::ErrorCode::NoHandshakeChannelBound`
  - `DsNIoFuncUart::ErrorCode::TransceiverBaudrateRange`
  - `DsNIoFuncUart::ErrorCode::TerminationMismatch`

  When an error occurs, the returned error code is also added to the ErrorList.

**Example**

```
pUartDriver->setDatabits(ErrorList, 7);
pUartDriver->setBaudrate(ErrorList, 115200);
pUartDriver->applySettings(ErrorList);      // The data bits and baudrate given
                                            // before are applied here
```

**Related topics**

References

DsNIoFuncUart::ErrorCode........................................................ 48

## DsCIoFuncUart::bindIoChannel

**Syntax**

```
void bindIoChannel(
   DsTErrorList   ErrorList,
   DsIoChannel*   pIoChannel,
   UInt32         LogicalChannelId,
   UInt32         ChannelRole)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To bind a single I/O channel to the driver.

**Description**

This function binds a single I/O channel to the driver. The driver can then perform I/O on the specific channel. At least one channel has to be bound to the driver.

If hardware flow control is desired, two consecutive channels must be bound. The `LogicalChannelId` and `ChannelRole` parameters specify additional settings for the channels. The following combinations are allowed:

- No hardware flow control allowed and no event generation desired: Bind one channel:
  - LogicalChannelId = DsNIoFuncUart::ChannelType::Data, ChannelRole = DsNIoChannelLink::Role::None
- No hardware flow control allowed but event generation desired: Bind one channel:
  - LogicalChannelId = DsNIoFuncUart::ChannelType::Data, ChannelRole = DsNIoChannelLink::Role::None
- Hardware flow control allowed and no event generation desired: Bind two consecutive channels:
  - LogicalChannelId = DsNIoFuncUart::ChannelType::Data, ChannelRole = DsNIoChannelLink::Role::None
  - LogicalChannelId = DsNIoFuncUart::ChannelType::Handshake, ChannelRole = DsNIoChannelLink::Role::None
- Hardware flow control allowed and event generation desired: Bind two consecutive channels:
  - LogicalChannelId = DsNIoFuncUart::ChannelType::Data, ChannelRole = DsNIoChannelLink::Role::None
  - LogicalChannelId = DsNIoFuncUart::ChannelType::Handshake, ChannelRole = DsNIoChannelLink::Role::None

For details on binding channels, refer to Creating and Adjusting the Source Code Files (Serial Communication) (ConfigurationDesk Custom I/O Function Implementation Guide 🕮).

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**pIoChannel** Specifies a pointer to channel object.

**LogicalChannelId** Specifies the type of the channel (for example, RX/TX or RTS/CTS), see DsNIoFuncUart::ChannelType on page 47.

**ChannelRole** Specifies the role of the channel if more than one channel is bound to the driver.

**Return value**

None

**Example**

The UART RS232 FlowControl Custom Function example shows how the function is used.

```
/** <!------------------------------------------------------------------------>
* Macro binds an I/O resource to a UART driver object (DO NOT CHANGE!)
*
* @parameters
*  @param P_DRIVER Pointer to the generated UART driver object (will be declared)
*  @param RESOURCE Name of the HW resource as declared in
  UART_RS232_FlowControl.xml
*  @param TYPE Channel type: 0: Use resource as data channel
*                            1: Use resource as handshake channel
* @param ROLE Role of channel:  0: Slave
*                               1: Master (Is able to generate Events etc.)
*
* @note
*  To use in create function
*
*<!------------------------------------------------------------------------>*/
#define UART_DRIVER_BIND_CHANNEL(P_DRIVER,RESOURCE,TYPE,ROLE) \
       P_DRIVER->bindIoChannel(ErrorList, \
       DsCIoChannel::getInstance(ErrorList, \
       DS_IO_MAP(g_ResourceMap, pBlockInstance->resIndex_##RESOURCE )) \
       ,TYPE ,ROLE );

// ...

/** <!------------------------------------------------------------------------>
* CustomIoFunction Create access point
*
* @description
*   This function is called for every created instance of the
*   UART_RS232_FlowControl CustomIoFunction. See the remarks below
*   for the detailed description.
*
* @parameters
*   @param ErrorList ErrorList to report errors
*   @param pBlockInstance Pointer to the BlockInstance buffer that is used
*      to handle different instances of the CustomIoFunction
*
*<!------------------------------------------------------------------------>*/
void ucf_Uart_Create(DsTErrorList ErrorList, ucf_UART_RS232_FlowControlStruct_T* pBlockInstance)
{
   // ...

   // 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)

   // ...
}
```

## DsCIoFuncUart::create

| | |
|---|---|
| **Syntax** | `static DsCIoFuncUart* create(`<br>`   DsTErrorList       ErrorList,`<br>`   DsCApplication*     pApplication)` |
| **Include file** | `DsIoFuncUart.h` |
| **Purpose** | To instantiate a DsCIoFuncUart driver object. |
| **Description** | This class method instantiates a DsCIoFuncUart driver object. Use the `bindIoChannel` method afterwards to assign a channel to the new driver object. For detailed information on how the `create` function is used, refer to Creating and Adjusting the Source Code Files (Serial Communication) (ConfigurationDesk Custom I/O Function Implementation Guide 🕮).<br><br>**Note**<br>When an error occurs, the following error codes can be added to the ErrorList:<br>▪ `DsNStdErrorCode::MemoryAllocation` |
| **Parameters** | **ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.<br><br>**pApplication** Specifies a pointer to the creating DsCApplication object. Normally, the DsCApplication object is provided by the application framework. See the `SimEngineApplGet()` call in the example below. |
| **Return value** | Pointer to the created driver object or NULL if the function failed. |
| **Example** | The UART RS232 FlowControl Custom Function example shows how the function is used. |

```
/** <!------------------------------------------------------------------------>
* 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_RS232_FlowControl.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 ; \


// ...


/** <!------------------------------------------------------------------------>
* CustomIoFunction Create access point
*
* @description
*   This function is called for every created instance of the UART_RS232_FlowControl
*   CustomIoFunction. See the remarks below for the detailed description.
*
* @parameters
*   @param ErrorList ErrorList to report errors
*   @param pBlockInstance Pointer to the BlockInstance buffer that is used
*      to handle different instances of the CustomIoFunction
*
*<!------------------------------------------------------------------------>*/
void ucf_Uart_Create(DsTErrorList ErrorList, ucf_UART_RS232_FlowControlStruct_T* pBlockInstance) {

   // ...

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

   // ...

}
```

**Related topics**

References

| | |
|---|---|
| DsCIoFuncUart::bindIoChannel | 63 |
| DsNStdErrorCode | 60 |

## DsCIoFuncUart::getChannelStatus

**Syntax**

```
UInt32 getChannelStatus   (
   DsTErrorList   ErrorList)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To get the status of the channel

**Description**

This method returns the current status of the channel, for example, you can use the method to retrieve the status of long running configuration sequences (baud rate setup etc.). The driver has to be started in order to get valid channel status information.

> **Tip**
>
> Only at the beginning of a task, new data and status information is received by the driver. It does therefore not make sense to poll for new data during a task.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**Return value**

See the error status flags of `DsNIoFuncUart::ChannelStatus`.

**Example**

```
// check if baud rate setup finished
if (    (pUartDriver->getChannelStatus(ErrorList) &
                  DsNIoFuncUart::ChannelStatus::BaudrateSetupPending) == 0 )
{
  // baudrate setup finished

  // ..
}
```

**Related topics**

References

DsNIoFuncUart::ChannelStatus........................................................................ 46

## DsCIoFuncUart::getErrorStatus

**Syntax**

```
UInt32 getErrorStatus   (
   DsTErrorList   ErrorList)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To get the error status, for example, FIFO overflow.

**Description**

This method returns the current error status of UART communication, for example, FIFO overflow. This is different from the error status of the driver object which is reported via the error list passed to the method calls. After reading the error status using this method, it is cleared internally.

**Tip**

Only at the beginning of a task, new data and status information is received by the driver. It does therefore not make sense to poll for new data during a task.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**Return value**

See the error status flags of `DsNIoFuncUart::ErrorStatus`.

**Example**

```
// check for receive FIFO overflow
if (pUartDriver->getErrorStatus(ErrorList) &
                        DsNIoFuncUart::ErrorStatus::RxFifoOverflow)
{
  // receive FIFO overflow occurred

  // ..
}
```

**Related topics**

References

DsNIoFuncUart::ErrorStatus........................................................ 50

## DsCIoFuncUart::getLineStatus

**Syntax**

```
UInt32 getLineStatus   (
   DsTErrorList   ErrorList)
```

| | |
|---|---|
| **Include file** | `DsIoFuncUart.h` |
| **Purpose** | To get the line status, for example, data ready, parity error, overrun error. |

**Description**

This method returns the current line status of the UART hardware. Flag constants are defined in `DsNIoFuncUart::UartLineStatus` to examine the individual bits of the status register. The driver has to be started in order to get valid channel status information. For details, refer to DsCIoFuncUart::start on page 93.

**Tip**

Only at the beginning of a task, new data and status information is received by the driver. It does therefore not make sense to poll for new data during a task.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**Return value**

See the line status bits of `DsNIoFuncUart::UartLineStatus`.

**Example**

```
// check for Parity Error
if (pUartDriver->getLineStatus(ErrorList) &
                              DsNIoFuncUart::UartStatus::LineParityError)
{
  // The parity error bit the in 16550 uart hardware device is set

  // ..
}
```

**Related topics**

References

DsNIoFuncUart::UartLineStatus ..... 58

## DsCIoFuncUart::getModemStatus

**Syntax**

```
UInt32 getModemStatus   (
   DsTErrorList   ErrorList)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To get the modem status, for example, CTS or DSR.

**Description**

This method returns the current modem status of the UART hardware. Flag constants are defined in `DsNIoFuncUart::UartModemStatus` to examine the individual bits of the status register. The driver has to be started in order to get valid channel status information. For details, refer to DsCIoFuncUart::start on page 93.

**Tip**

Only at the beginning of a task, new data and status information is received by the driver. It does therefore not make sense to poll for new data during a task.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**Return value**

See the modem status bits of `DsNIoFuncUart::UartModemStatus`.

**Example**

```
// check for CTS bit
if (pUartDriver->getModemStatus(ErrorList) &
                      DsNIoFuncUart::UartStatus::CTS)
{
  // CTS bit in 16550 uart hardware device is set

  // ..
}
```

**Related topics**

References

DsNIoFuncUart::UartModemStatus........................................................................... 59

## DsCIoFuncUart::getRxFifoLevel

**Syntax**

```
UInt32 getRxFifoLevel   (
   DsTErrorList   ErrorList)
```

**Include file**

`DsIoFuncUart.h`

**Purpose**

To get the number of bytes in the software receive FIFO.

**Description**

This method determines the number of bytes that are pending in the receive FIFO of the channel.

**Tip**

Only at the beginning of a task, new data and status information is received by the driver. It does therefore not make sense to poll for new data during a task.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**Return value**

Number of bytes that are pending in the receive FIFO of the channel.

**Example**

```
// Create variable for the FIFO level
UInt32 fifoLevel;

// Get the actual FIFO level
fifoLevel = pUartDriver->getRxFifoLevel(ErrorList);
```

## DsCIoFuncUart::getRxFifoSize

**Syntax**

```
UInt32 getRxFifoSize   (
   DsTErrorList   ErrorList)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To get the actual size of the software receive FIFO.

**Description**

This method determines the current size of the receive FIFO. The returned value represents the real size that was applied during the last call of the `DsCIoFuncUart::applySettings` method. This value might be different from the setting that was made with the `DsCIoFuncUart::setRxFifoSize` method before the last `applySettings` call. In addition, `setRxFifoSize` calls that were made without subsequently using `applySettings` have no effect on the real size of the receive FIFO.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**Return value**

The currently applied receive FIFO size.

**Example**

```
// Create variable for real receive FIFO size
UInt32 realRxFifoSize;

// Setup the receive FIFO
pUartDriver->setRxFifoSize(ErrorList, 256);
pUartDriver->setRxFifoMode(ErrorList, DsNIoFuncUart::RxFifoMode::OverwriteOldest);
pUartDriver->setRxFifoBlockSize(ErrorList, 5);
           // E. g. a protocol message is always
           // build of 5 bytes. If an overrun occurs
           // always sets of 5 bytes are overwritten
           // in order to keep the data consistent
pUartDriver->applySettings(ErrorList);

// Get the real receive FIFO size
realRxFifoSize = pUartDriver->getRxFifoSize(ErrorList);
                    // Now realRxFifoSize may be greater than 256!
```

**Related topics**

References

DsCIoFuncUart::applySettings........................................ 62
DsCIoFuncUart::setRxFifoSize........................................ 91

## DsCIoFuncUart::getTxFifoSpace

**Syntax**

```
UInt32 getTxFifoSpace   (
   DsTErrorList   ErrorList)
```

| | |
|---|---|
| **Include file** | `DsIoFuncUart.h` |
| **Purpose** | To get the space left in the transmit FIFO. |
| **Description** | This method determines the minimum number of bytes that fit into the transmit FIFO of the channel. |
| **Parameters** | **ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework. |
| **Return value** | Minimum number of bytes, that fit into the transmit FIFO of the channel. |

**Note**

The return value is the guaranteed minimum space left in the transmit FIFO. The real space left in the (hardware) transmit FIFO can be higher.

**Example**

```
// Create variable for the FIFO space
UInt32 fifoSpace;

// Get the minimum transmit FIFO space
fifoSpace = pUartDriver->getTxFifoSpace(ErrorList);
```

## DsCIoFuncUart::receive

**Syntax**

```
DsTError receive   (
   DsTErrorList   ErrorList,
   UInt32         MaxBytes,
   UInt8*         pData,
   UInt32*        pBytesReceived)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose** To check for received user data.

**Description**

This method checks for received user data. Data bytes pending in the software receive FIFO are copied to the specified data buffer. If more bytes are pending than the `MaxBytes` parameter specifies, only the number of `MaxBytes` are copied. Additional bytes remain in the FIFO and can be received later. The `pBytesReceived` parameter is used to notify the caller about the copied (received) bytes.

**Tip**

Only at the beginning of a task, new data and status information is received by the driver. It does therefore not make sense to poll for new data during a task.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**MaxBytes** Specifies the maximum number of bytes that can be copied to the data buffer.

**pData** Specifies a pointer to the buffer for the received data. The size must be at least `MaxBytes`.

**pBytesReceived** Specifies a pointer to a UInt32 variable that stores the number of bytes copied to the buffer.

**Return value**

Always returns 0 (no error).

**Example**

```
// create Buffer for sending data
UInt8 Buffer[512];
UInt32 MaxBytes=512;
UInt32 BytesReceived;

// ...

// Check for received user data
pUartDriver->receive(ErrorList,MaxBytes,Buffer,&BytesReceived);

// Something received?
if(BytesReceived)
{
  // Process received data

  // ...
}
```

## DsCIoFuncUart::receiveTerm

**Syntax**

```
DsTError receiveTerm   (
   DsTErrorList   ErrorList,
   UInt32         MaxBytes,
   UInt8*         pData,
   UInt32*        pBytesReceived,
   UInt8          Term)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To get received user data until a termination character is detected.

**Description**

This method checks for received user data. Data bytes pending in the software receive FIFO are copied to the specified data buffer. The copy process is terminated if a specified termination character was detected, if the number of `MaxBytes` data bytes have been copied or if the receive FIFO is empty. Additional bytes remain in the FIFO and can be received later. The `pBytesReceived` parameter is used to notify the caller about the copied (received) bytes.

**Tip**

Only at the beginning of a task, new data and status information is received by the driver. It does therefore not make sense to poll for new data during a task.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**MaxBytes** Specifies the maximum number of bytes that can be copied to the data buffer.

**pData** Specifies a pointer to the buffer for the received data. The size must be at least `MaxBytes`.

**pBytesReceived** Specifies a pointer to a UInt32 variable that stores the number of bytes copied to the buffer.

**Term** Specifies a termination character. The copy process is terminated if a data byte is equal to this value. The termination character itself is also copied.

**Return value**

Always returns 0 (no error).

**Example**

```
// create Buffer for sending data
UInt8 Buffer[512];
UInt32 MaxBytes=512;
UInt32 BytesReceived;

// ...

// Check for received user data string terminated by a line feed.
pUartDriver->receiveTerm(ErrorList,MaxBytes,Buffer,&BytesReceived,'\n');

// If a '\n' was detected, additional bytes remain in the
// receive FIFO. But be careful: Even if no '\n' was detected,
// pending data is copied to the Buffer!

// Something received?
if(BytesReceived)
{
  // Process received data

  // ...
}
```

## DsCIoFuncUart::resetRxFifo

**Syntax**

```
DsTError resetRxFifo  (
   DsTErrorList   ErrorList)
```

**Include file**

`DsIoFuncUart.h`

**Purpose**

To clear the receive FIFO.

**Description**

This method clears the receive FIFO. All the received bytes in the buffers are deleted.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**Return value**

Always returns 0 (No Error).

**Example**

```
pUartDriver->resetRxFifo(ErrorList);
```

## DsCIoFuncUart::send

**Syntax**

```
DsTError send   (
   DsTErrorList   ErrorList,
   UInt32         BytesToSend,
   UInt8*         pData,
   UInt32*        pBytesSent)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To send data.

**Description**

This method is used to send user data. Depending on the internal state of the driver and the length of the user data block to be sent, it can happen that not all the bytes are sent. Use the `pBytesSent` parameter to determine the number of bytes really sent.

**Note**

The maximum number of bytes send per call is 200 (for UART 5: 16). Use additional `send()` calls (in a loop) to send larger blocks.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**BytesToSend** Specifies the number of bytes in the data buffer to be sent.

**pData** Specifies a pointer to the data buffer that holds the bytes to be sent.

**pBytesSent** Specifies a pointer to a UInt32 variable that stores the number of bytes really sent by this method. This number can be less than the value specified in the `BytesToSend` parameter.

**Return value**

Always returns 0 (no error).

**Example**

```
// create Buffer for sending data
UInt8 Buffer[512];
UInt8 *pBuffer;
UInt32 BytesToSend=512;
UInt32 BytesSent;

// ... (some additional code, e. g. filling Buffer etc.)

// Check if Transmit FIFO has enough space for data
if (pUartDriver->getTxFifoSpace(ErrorList) >= BytesToSend)
{
   // Initialize data pointer
   pBuffer = Buffer;

   // loop to send all bytes
   while(BytesToSend > 0)
   {
      // Try to send as much bytes as necessary
      pUartDriver->send(ErrorList,BytesToSend,Buffer,&BytesSent);

      // Calculate number of remaining bytes
      BytesToSend -= BytesSent;

      // Calculate pointer to remaining data
      pBuffer+= BytesSent;
   }
}
```

**Related topics**

References

DsCIoFuncUart::start........ 93
DsCIoFuncUart::stop........ 94

## DsCIoFuncUart::setBaudrate

**Syntax**

```
DsTError setBaudrate  (
   DsTErrorList   ErrorList,
   UInt32         Baudrate)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To set the baud rate for the channel.

**Description**

This method sets the baud rate of the assigned channel. Use the `DsCIoFuncUart::applySettings` method to activate the new setting. The baud rate setup sequence may need several milliseconds to be completed. Use the `DsCIOfuncUart::getChannelStatus` method to retrieve the current state of the sequence.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**Baudrate** Specifies the baud rate.

**Return value**

- 0 if no error occurred.
- Otherwise error code as specified in `DsNIoFuncUart::ErrorCode`:
  - `DsNIoFuncUart::ErrorCode::UartBaudrateRange`

  When an error occurs, the returned error code is also added to the ErrorList.

**Example**

```
pUartDriver->setBaudrate(ErrorList, 115200);
pUartDriver->applySettings(ErrorList);
```

**Related topics**

Basics

Defining the Configuration Properties (Serial Communication) (ConfigurationDesk Custom I/O Function Implementation Guide 🕮)

References

Baud Rates for Onboard UART of SCALEXIO Processing Hardware........ 80
DsCIoFuncUart::applySettings........ 62
DsCIoFuncUart::getChannelStatus........ 67
DsNIoFuncUart::ErrorCode........ 48

## Baud Rates for Onboard UART of SCALEXIO Processing Hardware

**Setting the baud rates for processing hardware onboard UART**

The onboard UART of SCALEXIO processing hardware operates with a fixed frequency. To set the baud rate, a divisor is used. As divisors are integer values, you cannot realize any desired value for the baud rate. There are quantization effects with rough steps, especially for high baud rates.

**Calculating the real baud rate**

You can calculate the real baud rate by calculating the divisor for the desired baud rate according to the fixed frequency of the hardware. The divisor must be an integer value.

```
Divisor = round_down(( (UART_ClockFrequency) / (16 *
Baudrate_Desired ) + 0.5)
```

Use the following formula to calculate the real baud rate:

```
Baudrate_Real = UART_ClockFrequency / (16 * Divisor)
```

The `UART-ClockFrequency` value depends on the hardware:
- SCALEXIO Processing Unit: 1,843,200.0 Hz
- DS6001 Processor Board: 14,705,882.35294 Hz

There are quantization effects because of the float to integer conversion. Refer to the following table for examples of baud rates depending on divisors for the respective hardware.

| Baudrate_Desired | SCALEXIO Processing Unit | | DS6001 Processor Board | |
|---|---|---|---|---|
| | Baudrate_Real | Divisor | Baudrate_Real | Divisor |
| 1,200 | 1,200 | 96 | 1,199.9 | 766 |
| 2,400 | 2,400 | 48 | 2,399.8 | 383 |
| 4,800 | 4,800 | 24 | 4,812.1 | 191 |
| 9,600 | 9,600 | 12 | 9,574.1 | 96 |
| 19,200 | 19,200 | 6 | 19,148.3 | 48 |
| 20,000 | 23,040 | 5 | 20,424.8 | 45 |
| 56,000 | 57,600 | 2 | 57,444.9 | 16 |
| 115,200 | 115,200 | 1 | 114,889.7 | 8 |

**Related topics**

Examples

Example of UART Serial Communication Using Onboard UART of SCALEXIO Processing Hardware........................................................................................................... 17

References

DsCIoFuncUart::setBaudrate........................................................................................................... 79

## DsCIoFuncUart::setDatabits

**Syntax**

```
DsTError setDatabits  (
   DsTErrorList   ErrorList,
   UInt32         Databits)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To set the number of data bits.

**Description**

This method sets the number of data bits of the data frame. Use the `DsCIoFuncUart::applySettings` method to activate the new setting.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**Databits** Specifies the number of data bits, see DsNIoFuncUart::Databits on page 48.

**Return value**

- 0 if no error occurred.
- Otherwise error code as specified in `DsNIoFuncUart::ErrorCode`:
  - `DsNIoFuncUart::ErrorCode::DatabitsRange`

  When an error occurs, the returned error code is also added to the ErrorList.

**Example**

```
pUartDriver->setDatabits(ErrorList, 7);
pUartDriver->applySettings(ErrorList);
```

**Related topics**

References

DsCIoFuncUart::applySettings........................................................................................... 62
DsNIoFuncUart::ErrorCode................................................................................................ 48

## DsCIoFuncUart::setFlowControlMode

**Syntax**

```
DsTError setFlowControlMode   (
   DsTErrorList   ErrorList,
   UInt32         FlowControlMode)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To set the flow control mode.

**Description**

This function sets the flow control mode of the driver object. Use the `DsCIoFuncUart::applySettings` method to activate the new setting.

> **Note**
>
> To use hardware flow control with the DS2671 Bus Board, two channels have to be bundled using the `DsCIoFuncUart::bindIoChannel` method.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**FlowControlMode** Specifies the flow control mode, see DsNIoFuncUart::FlowControlMode on page 52.

**Return value**

- 0 if no error occurred.
- Otherwise error code as specified in `DsNIoFuncUart::ErrorCode`:
  - `DsNIoFuncUart::ErrorCode::FlowControlModeRange`

  When an error occurs, the returned error code is also added to the ErrorList.

**Example**

```
pUartDriver->setFlowControlMode(ErrorList,
DsNIoFuncUart::FlowControlMode::Hardware);
pUartDriver->applySettings(ErrorList);
```

**Related topics**

Basics

Creating and Adjusting the Source Code Files (Serial Communication) (ConfigurationDesk Custom I/O Function Implementation Guide 📖)

References

DsCIoFuncUart::applySettings........................................ 62
DsCIoFuncUart::bindIoChannel........................................ 63
DsNIoFuncUart::ErrorCode........................................ 48

## DsCIoFuncUart::setStopbits

**Syntax**

```
DsTError setStopbits  (
   DsTErrorList   ErrorList,
   UInt32         Stopbits)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To set the number of stop bits in a data frame.

**Description**

This method sets the number of stop bits at the end of a data frame. Use the `DsCIoFuncUart::applySettings` method to activate the new setting.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**Stopbits** Specifies the number of stop bits, see DsNIoFuncUart::Stopbits on page 56).

**Return value**

- 0 if no error occurred.
- Otherwise error code as specified in `DsNIoFuncUart::ErrorCode`:
  - `DsNIoFuncUart::ErrorCode::StopbitsRange`

  When an error occurs, the returned error code is also added to the ErrorList.

**Example**

```
pUartDriver->setStopbits(ErrorList, DsNIoFuncUart::Stopbits::Two);
pUartDriver->applySettings(ErrorList);
```

**Related topics**

References

DsCIoFuncUart::applySettings.......... 62
DsNIoFuncUart::ErrorCode.......... 48

## DsCIoFuncUart::setParity

**Syntax**

```
DsTError setParity   (
    DsTErrorList   ErrorList,
    UInt32         Parity)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To set the parity for the data frames.

| | |
|---|---|
| **Description** | This function sets the calculation method for generating a parity bit in a data frame. Use the `DsCIoFuncUart::applySettings` method to activate the new setting. |
| **Parameters** | **ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.<br><br>**Parity** Specifies the calculation method for the parity bit, see DsNIoFuncUart::Parity on page 53. |
| **Return value** | ▪ 0 if no error occurred.<br>▪ Otherwise error code as specified in `DsNIoFuncUart::ErrorCode`:<br>  ▪ `DsNIoFuncUart::ErrorCode::ParityRange`<br>When an error occurs, the returned error code is also added to the ErrorList. |

**Example**

```
pUartDriver->setParity(ErrorList, DsNIoFuncUart::Parity::Even);
pUartDriver->applySettings(ErrorList);
```

**Related topics**

References

DsCIoFuncUart::applySettings........ 62
DsNIoFuncUart::ErrorCode........ 48

## DsCIoFuncUart::setTermination

**Syntax**

```
DsTError setTermination   (
   DsTErrorList   ErrorList,
   UInt32         Termination)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To select a termination for the channel.

**Description**

This function can be used to select a termination for the channel. Use the `DsCIoFuncUart::applySettings` method to activate the new setting. The termination setup sequence may need several milliseconds to be completed. Use the `DsCIOfuncUart::getChannelStatus` method to retrieve the current state of the sequence.

**Note**

The allowed terminations depend on the selected transceiver. See also DsCIoFuncUart::setTransceiver on page 86 and DsCIoFuncUart::applySettings on page 62.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**Termination** Specifies the termination type, see DsNIoFuncUart::Termination on page 56.

**Return value**

- 0 if no error occurred.
- Otherwise error code as specified in `DsNIoFuncUart::ErrorCode`:
  - `DsNIoFuncUart::ErrorCode::TerminationRange`
  - `DsNIoFuncUart::ErrorCode::TerminationMismatch`

  When an error occurs, the returned error code is also added to the ErrorList.

**Example**

```
pUartDriver->setTermination(ErrorList, DsNIoFuncUart::Termination::Parallel_AB);
pUartDriver->applySettings(ErrorList);
```

**Related topics**

References

DsCIoFuncUart::applySettings.................................................................62
DsCIoFuncUart::getChannelStatus.............................................................67
DsNIoFuncUart::ErrorCode.......................................................................48

## DsCIoFuncUart::setTransceiver

**Syntax**

```
DsTError setTransceiver   (
   DsTErrorList   ErrorList,
   UInt32         Transceiver)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To select a transceiver for a channel.

**Description**

This function can be used to select a transceiver for the channel. Use the `DsCIoFuncUart::applySettings` method to activate the new setting. The transceiver setup sequence may need several milliseconds to be completed. Use the `DsCIOfuncUart::getChannelStatus` method to retrieve the current state of the sequence.

**Note**

The transceiver used can affect the allowed baud rate.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**Transceiver** Specifies the transceiver type, see DsNIoFuncUart::Transceiver on page 57.

**Return value**

- 0 if no error occurred.
- Otherwise error code as specified in `DsNIoFuncUart::ErrorCode`:
  - `DsNIoFuncUart::ErrorCode::TransceiverRange`

  When an error occurs, the returned error code is also added to the ErrorList.

**Example**

```
pUartDriver->setTransceiver(ErrorList, DsNIoFuncUart::Transceiver::RS232);
pUartDriver->applySettings(ErrorList);
```

**Related topics**

References

DsCIoFuncUart::applySettings........ 62
DsCIoFuncUart::getChannelStatus........ 67
DsNIoFuncUart::ErrorCode........ 48

## DsCIoFuncUart::setTriggerEnable

**Syntax**

```
DsTError setTriggerEnable   (
   DsTErrorList   ErrorList,
   UInt32         EventId,
   UInt8          Enable)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To enable or disable individual trigger sources.

**Description**

This method enables or disables the trigger generation for trigger source. The trigger source is specified by the `EventId` parameter. In order to trigger tasks, the driver object has to be bound to the task as a trigger first. For details, refer to DsNIoFuncUart::Event on page 50. If the driver object was bound to a task as a trigger and the trigger source is enabled, the task will be triggered if the specified event occurs. By default, the specified trigger source is enabled.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**EventId** Specifies the Id of the event to be enabled or disabled as a trigger source for a event-triggered task, see DsNIoFuncUart::Event on page 50.

**Enable** Specifies whether the trigger source should be enabled (value not zero) or disabled (value zero).

**Return value**

- 0 if no error occurred.
- Otherwise error code as specified in `DsNIoFuncUart::ErrorCode`:
  - `DsNIoFuncUart::ErrorCode::EventIdMaxRange`

**Example**

```
// During run time: Disable the individual trigger source
pUartDrv->setTriggerEnable(ErrorList,DsNIoFuncUart::Event::LineStatusChanged,0);
```

**Related topics**

References

DsNIoFuncUart::ErrorCode........................................................................................................ 48

## DsCIoFuncUart::setRxFifoBlockSize

**Syntax**

```
DsTError setRxFifoBlockSize  (
   DsTErrorList   ErrorList,
   UInt32         BlockSize)
```

| | |
|---|---|
| **Include file** | `DsIoFuncUart.h` |
| **Purpose** | To set the block size for a receive FIFO in overwrite mode. |
| **Description** | If the receive FIFO mode was set to `DsNIoFuncUart::RxFifoMode::OverwriteOldest` (see DsNIoFuncUart::RxFifoMode on page 54) with the `setRxFifoMode` method, this method can be used to specify the block size for this operation. The block size gives the number of bytes that are deleted from the FIFO if the FIFO overruns. Use the `applySettings` method to activate the new setting. |
| **Parameters** | **ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.<br>**BlockSize** Specifies the block size within the range: 1 .. (receive FIFO size - 1), see DsNIoFuncUart::RxFifoBlockSize on page 53. |
| **Return value** | ▪ 0 if no error occurred.<br>▪ Otherwise error code as specified in `DsNIoFuncUart::ErrorCode`:<br>  ▪ `DsNIoFuncUart::ErrorCode::RxFifoBlockSizeMinRange`<br>  ▪ `DsNIoFuncUart::ErrorCode::RxFifoBlockSizeMaxRange`<br>When an error occurs, the returned error code is also added to the ErrorList. |

**Example**

The following example illustrates the function: The FIFO is set to the `OverwiteOldest` mode and the block size is set to 4 because always frames of 4 bytes are expected. Only 2 bytes of free space are left in the FIFO and a new data frame with 4 bytes is to be received. The FIFO is full after the reception of the first 2 bytes of the frame. Because of the block size of 4, the oldest 4 bytes are deleted from the FIFO, not only one byte or two bytes. Due to this, the oldest frame is deleted completely and not only a part of the frame. Now, the last two bytes of the new frame are received. They fit into the FIFO and 2 additional bytes of free space are left afterwards. With this method, misalignments in the receive FIFO are avoided, if only frames of 4 bytes are received.

```
pUartDriver->setRxFifoSize(ErrorList, DsNIoFuncUart::RxFifoSize::Default);
pUartDriver->setRxFifoMode(ErrorList, DsNIoFuncUart::RxFifoMode::OverwriteOldest);
pUartDriver->setRxFifoBlockSize(ErrorList, 4);
         // E. g. a protocol message is always
         // build of 4 bytes. If an overrun occurs
         // always sets of 4 bytes are overwritten
         // in order to keep the data consistent
pUartDriver->applySettings(ErrorList);
```

**Related topics**

References

DsCIoFuncUart::applySettings........ 62
DsCIoFuncUart::setRxFifoMode........ 90
DsNIoFuncUart::ErrorCode........ 48

## DsCIoFuncUart::setRxFifoMode

**Syntax**

```
DsTError setRxFifoMode  (
   DsTErrorList   ErrorList,
   UInt32         RxFifoMode)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To set the mode of the receive FIFO if an overrun occurs.

**Description**

This method specifies the behavior of the receive FIFO if an overrun occurs. The following modes are provided:

- `DsNIoFuncUart::RxFifoMode::OverwriteOldest` mode

  If the FIFO overruns, the oldest data is deleted. The number of bytes which are delete are specified in a block size. The block size for the overwrite operation can be specified with the `setRxFifoBlockSize` method.
- `DsNIoFuncUart::RxFifoMode::DiscardNew` mode

  If the FIFO overruns, all additional received data is discarded. The block size specified with the `setRxFifoBlockSize` method has no effect in this mode.

Use the `applySettings` method to activate the new setting.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**RxFifoMode** Specifies the overwrite mode, see DsNIoFuncUart::RxFifoMode on page 54.

**Return value**

- 0 if no error occurred.
- Otherwise error code as specified in `DsNIoFuncUart::ErrorCode`:
  - `DsNIoFuncUart::ErrorCode::RxFifoModeRange`

  When an error occurs, the returned error code is also added to the ErrorList.

**Example**

```
pUartDriver->setRxFifoSize(ErrorList, DsNIoFuncUart::RxFifoSize::Default);
pUartDriver->setRxFifoMode(ErrorList, DsNIoFuncUart::RxFifoMode::DiscardNew);
pUartDriver->applySettings(ErrorList);
```

**Related topics**

References

| | |
|---|---|
| DsCIoFuncUart::applySettings | 62 |
| DsCIoFuncUart::setRxFifoBlockSize | 88 |
| DsNIoFuncUart::ErrorCode | 48 |

## DsCIoFuncUart::setRxFifoSize

**Syntax**

```
DsTError setRxFifoSize (
   DsTErrorList   ErrorList,
   UInt32         RxFifoSize)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To set the size of the receive FIFO.

**Description**

This method sets the size of the receive FIFO. This is a software FIFO and therefore independent of any hardware FIFO. Use the `applySettings` method to activate the new setting. If the size of an existing receive FIFO is changed with this method, the FIFO is cleared during the next call of the `applySettings` method. Because of the internal realization of the FIFO, the maximum number of bytes that can be stored is equal to (Fifo size - 1). In some situations the FIFO size can be automatically increased above the specified value during the next call of the `applySettings` method, for example, if `OverwiteOldest` mode is used.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**RxFifoSize** Specifies the size of the receive FIFO, see DsNIoFuncUart::RxFifoSize on page 54.

**Return value**

0 if no error occurred.

Otherwise error code as specified in `DsNIoFuncUart::ErrorCode`:

- `DsNIoFuncUart::ErrorCode::RxFifoSizeRange`

When an error occurs, the returned error code is also added to the ErrorList.

**Example**

```
pUartDriver->setRxFifoSize(ErrorList, DsNIoFuncUart::RxFifoSize::Default);
pUartDriver->applySettings(ErrorList);
```

**Related topics**

References

DsCIoFuncUart::applySettings.......... 62
DsCIoFuncUart::setRxFifoMode.......... 90
DsNIoFuncUart::ErrorCode.......... 48

## DsCIoFuncUart::setRxFifoTriggerLevel

**Syntax**

```
DsTError setRxFifoTriggerLevel   (
    DsTErrorList   ErrorList,
    UInt32         TriggerLevel)
```

**Include file**

```
DsIoFuncUart.h
```

**Purpose**

To set the trigger level for the software receive FIFO.

**Description**

This method sets the trigger level for the receive FIFO. If the receive FIFO contains at least this number of bytes, an RxFifoLevelReached event is generated that may trigger an event-triggered task (see DsNIoFuncUart::Event on page 50 and DsCIoFuncUart::setTriggerEnable on page 87). Use the `applySettings` method to activate the new setting.

**Note**

- If you use a driver instance in several tasks, for example, a periodic task and an event-driven task, you may never reach the trigger level because both tasks contend for the data.
- If you specify a trigger level above 50 bytes, you need a periodic task instead of an event-driven task. Otherwise a buffer overflow can occur.
- For the UART 5 channel type, only four constant values are available. Refer to DsNIoFuncUart::RxFifoTriggerLevel on page 55.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**TriggerLevel** Specifies the trigger level within the range 1 .. (receive FIFO size - 1), see DsNIoFuncUart::RxFifoTriggerLevel on page 55.

**Return value**

- 0 if no error occurred.
- Otherwise error code as specified in `DsNIoFuncUart::ErrorCode`:
  - `DsNIoFuncUart::ErrorCode::RxFifoTriggerLevelMinRange`
  - `DsNIoFuncUart::ErrorCode::RxFifoTriggerLevelMaxRange`

  When an error occurs, the returned error code is also added to the ErrorList.

**Example**

```
pUartDriver->setRxFifoSize(ErrorList, DsNIoFuncUart::RxFifoSize::Default);
pUartDriver->setRxFifoTriggerLevel(ErrorList, 2);
                // If a call back function is assigned
                // to the receive trigger level event, it is
                // invoked, if at least 2 bytes are pending in the
                // receive fifo.
pUartDriver->applySettings(ErrorList);
```

**Related topics**

References

| | |
|---|---|
| DsCIoFuncUart::applySettings | 62 |
| DsNIoFuncUart::ErrorCode | 48 |
| DsNIoFuncUart::RxFifoTriggerLevel | 55 |

## DsCIoFuncUart::start

**Syntax**

```
DsTError start   (
   DsTErrorList   ErrorList)
```

| | |
|---|---|
| **Include file** | `DsIoFuncUart.h` |
| **Purpose** | To start UART operation. |
| **Description** | This method starts the operation of the UART, i. e. sending, receiving and interrupt handling. The `getChannelStatus` method provides valid information only if the driver is started. Internal error flags are cleared during the execution of the `start` method. |
| **Parameters** | **ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework. |
| **Return value** | Always returns 0 (no error). |

**Example**

```
// On model start
pUartDriver->start(ErrorList);      // Start the operation of the uart
```

**Related topics**

References

DsCIoFuncUart::getChannelStatus........................................................................................... 67

## DsCIoFuncUart::stop

**Syntax**

```
DsTError stop   (
   DsTErrorList   ErrorList)
```

| | |
|---|---|
| **Include file** | `DsIoFuncUart.h` |
| **Purpose** | To stop UART operation. |
| **Description** | This method stops the operation of the UART, i. e. sending, receiving and interrupt handling. Data, that is still pending in the transmit FIFO will be sent, even if the `stop` method is called. After a call of the `stop` method, new data |

cannot be added to the transmit FIFO by means of the `send` method until the `start` method is called again.

**Parameters**

**ErrorList** Specifies a `DsTErrorList` object that is used for error handling. If an error occurs, the corresponding error code is added to the `ErrorList`. Normally, the `DsTErrorList` object is provided by the application framework.

**Return value**

Always returns 0 (no error).

**Example**

```
// On model stop
pUartDriver->stop(ErrorList);       // Stop the uart operation
                                    // Pending send operations are canceled,
                                    // receive FIFO is cleared
```

**Related topics**

References

DsCIoFuncUart::send.......... 78
DsCIoFuncUart::start.......... 93

# 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........98
B........98
C........101
D........104
E........105
F........107
G........108
H........108
I........109
L........110
M........111
N........114
O........114
P........114
R........116
S........117
T........118
U........119

V........................................................................................................120
W.......................................................................................................120
X........................................................................................................121

## 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.

**A**

applySettings 62

**B**

Baudrate (DsNIoFuncUart) 46
bindIoChannel 63
building signal chain
  for communication via serial interface 9

**C**

ChannelStatus (DsNIoFuncUart) 46
ChannelType (DsNIoFuncUart) 47
Common Program Data folder 8, 102
communication via serial interface
  building signal chain 9
create 66

**D**

Databits (DsNIoFuncUart) 48
Documents folder 8, 104
DsCIoFuncUart
  error codes 48
  function overview 60
DsCIoFuncUart class 44
DsNIoFuncUart
  namespace 44
DsNStdErrorCode 60

**E**

error codes for DsCIoFuncUart module 48
ErrorCode (DsNIoFuncUart) 48
ErrorStatus (DsNIoFuncUart) 50
Event (DsNIoFuncUart) 50

**F**

FlowControlMode (DsNIoFuncUart) 52
function overview
  DsCIoFuncUart 60

**G**

getChannelStatus 67
getErrorStatus 68
getLineStatus 69
getModemStatus 71
getRxFifoLevel 72
getRxFifoSize 72
getTxFifoSpace 73

**L**

Local Program Data folder 8, 110

**N**

namespace
  DsNIoFuncUart 44

**P**

Parity (DsNIoFuncUart) 53

**R**

receive 74
receiveTerm 76
resetRxFifo 77
RxFifoBlockSize (DsNIoFuncUart) 53
RxFifoMode (DsNIoFuncUart) 54
RxFifoSize (DsNIoFuncUart) 54
RxFifoTriggerLevel (DsNIoFuncUart) 55

**S**

send 78
serial communication
  data flow 11
setBaudrate 79
setDatabits 81
setFlowControlMode 82
setParity 84
setRxFifoBlockSize 88
setRxFifoMode 90
setRxFifoSize 91
setRxFifoTriggerLevel 92
setStopbits 83
setTermination 85
setTransceiver 86
setTriggerEnable 87
start 93
stop 94
Stopbits (DsNIoFuncUart) 56

**T**

Termination (DsNIoFuncUart) 56
Transceiver (DsNIoFuncUart) 57

**U**

UART
  data flow 11
UART core 11
UART driver 11
UART serial communication 9
UartLineStatus (DsNIoFuncUart) 58
UartModemStatus (DsNIoFuncUart) 59