SIEMENS

Realtime Information Backbone (RIB)

Programming Manual

S7-1500/ET 200MP Documentation Guide	1
Introduction	2
Safety information	3
Product overview / Software description	4
Software installation and uninstallation	5
RIB_Application	6
RIB Support Library	7
Programming using the SFC ODKP_ISC (SFC65490)	8
Commissioning (software)	9
Examples and tables	A
List of abbreviations	B

05/2023

A5E52046002-AB

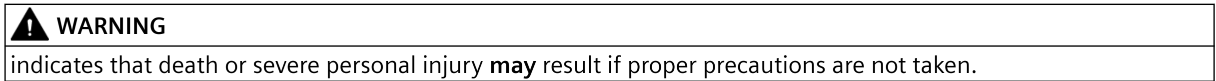
Legal information

Warning notice system

This manual contains notices you have to observe in order to ensure your personal safety, as well as to prevent damage to property. The notices referring to your personal safety are highlighted in the manual by a safety alert symbol, notices referring only to property damage have no safety alert symbol. These notices shown below are graded according to the degree of danger.

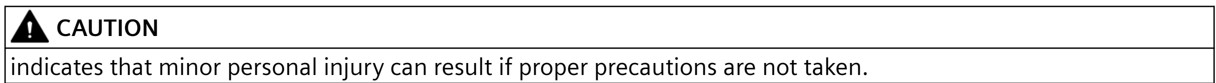
DANGER

indicates that death or severe personal injury **will** result if proper precautions are not taken.



WARNING

indicates that death or severe personal injury **may** result if proper precautions are not taken.



CAUTION

indicates that minor personal injury can result if proper precautions are not taken.

NOTICE

indicates that property damage can result if proper precautions are not taken.

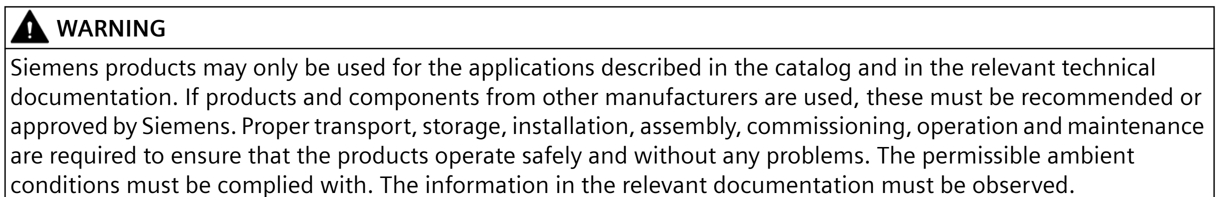
If more than one degree of danger is present, the warning notice representing the highest degree of danger will be used. A notice warning of injury to persons with a safety alert symbol may also include a warning relating to property damage.

Qualified Personnel

The product/system described in this documentation may be operated only by **personnel qualified** for the specific task in accordance with the relevant documentation, in particular its warning notices and safety instructions. Qualified personnel are those who, based on their training and experience, are capable of identifying risks and avoiding potential hazards when working with these products/systems.

Proper use of Siemens products

Note the following:



WARNING

Siemens products may only be used for the applications described in the catalog and in the relevant technical documentation. If products and components from other manufacturers are used, these must be recommended or approved by Siemens. Proper transport, storage, installation, assembly, commissioning, operation and maintenance are required to ensure that the products operate safely and without any problems. The permissible ambient conditions must be complied with. The information in the relevant documentation must be observed.

Trademarks

All names identified by ® are registered trademarks of Siemens AG. The remaining trademarks in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owner.

Disclaimer of Liability

We have reviewed the contents of this publication to ensure consistency with the hardware and software described. Since variance cannot be precluded entirely, we cannot guarantee full consistency. However, the information in this publication is reviewed regularly and any necessary corrections are included in subsequent editions.

Siemens AG
Digital Industries
Postfach 48 48
90026 NÜRNBERG
GERMANY

A5E52046002-AB
Ⓟ 05/2023 Subject to change

Copyright © Siemens AG 2021 - 2023.
All rights reserved

Table of contents

1 S7-1500/ET 200MP Documentation Guide 5
1.1 S7-1500 / ET 200MP Documentation Guide 5
1.2 SIMATIC Technical Documentation 6
1.3 Tool support 8
2 Introduction 9
2.1 Operating instructions guide 9
3 Safety information 11
3.1 Security information 11
3.2 Open-source software 11
4 Product overview / Software description 13
4.1 Integration of RIB 13
4.2 Realtime data exchange 15
4.2.1 Realtime data exchange concept 15
4.2.2 Adjustments for correct realtime behavior 16
4.2.3 Lifetime buffer 18
4.2.4 Writing data to lifetime buffer 20
4.2.5 Reading data from lifetime buffer 21
4.2.6 Buffer example 22
5 Software installation and uninstallation 25
5.1 Installing RIB on CPU 1518(F)-4 PN/DP MFP 25
5.2 Installing RIB on Software Controller 25
5.3 Compatibility rules 27
5.4 Uninstalling RIB from Software Controller 28
6 RIB_Application 31
6.1 RIB_Application 31
6.2 Diagnostic data 33
6.3 General response 34
6.4 Global RIB environment 35
6.5 Connecting client applications 35
6.6 Data processing and symbol matching 38
6.7 Connection response 39
6.8 Disconnecting client applications 41
6.9 Programming example 44

7 **RIB Support Library** ... **47**

7.1 Introduction ... 47

7.2 Connecting a client to RIB ... 48

7.3 Defining symbols ... 49

7.4 Activating client ... 51

7.5 Reading data from lifetime buffer ... 52

7.6 Writing data to lifetime buffer ... 57

7.7 Deactivating client ... 59

7.8 Obtaining a string for RibReturnCode ... 60

7.9 Registering and unregistering signal callback functions ... 61

8 **Programming using the SFC ODKP_ISC (SFC65490)** ... **63**

8.1 Configuration ... 63

8.2 "OP_INIT" mode ... 64
8.2.1 "Init" operation ... 64
8.2.2 ODKP_ISC "Initialize" mode parameters ... 66

8.3 "OP_CONNECT" mode ... 69
8.3.1 Connecting to RIB ... 69

8.4 "OP_READ" mode ... 71
8.4.1 Reading data from Linux applications ... 71

8.5 "OP_WRITE" mode ... 73
8.5.1 Writing data from Linux applications ... 73

8.6 "OP_DISCONNECT" mode ... 75
8.6.1 Disconnecting from RIB ... 75

9 **Commissioning (software)** ... **77**

9.1 Cleaning lifetime buffer ... 77

A **Examples and tables** ... **79**

A.1 Example project ... 79

A.2 Application examples ... 86

A.3 Allowed data types ... 92

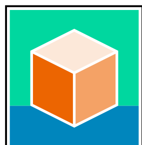
A.4 Example configuration string ... 92

A.5 RibReturnCode ... 94

B **List of abbreviations** ... **97**

1 S7-1500/ET 200MP Documentation Guide

1.1 S7-1500 / ET 200MP Documentation Guide



The documentation for the SIMATIC S7-1500 automation system and the ET 200MP distributed I/O system is arranged into three areas.

This arrangement enables you to access the specific content you require. Changes and supplements to the manuals are documented in a Product Information.

You can download the documentation free of charge from the Internet (https://support.industry.siemens.com/cs/ww/en/view/109742691).

Basic information

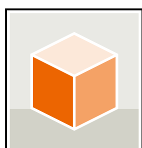
The System Manual and Getting Started describe in detail the configuration, installation, wiring and commissioning of the SIMATIC S7-1500 and ET 200MP systems.

The STEP 7 online help supports you in the configuration and programming.

Examples:

- Getting Started S7-1500
- S7-1500/ET 200MP System Manual
- Online help TIA Portal

Device information

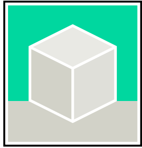


Equipment manuals contain a compact description of the module-specific information, such as properties, wiring diagrams, characteristics and technical specifications.

Examples:

- Equipment Manuals CPUs
- Equipment Manuals Interface Modules
- Equipment Manuals Digital Modules
- Equipment Manuals Analog Modules
- Equipment Manuals Communications Modules
- Equipment Manuals Technology Modules
- Equipment Manuals Power Supply Modules

General information



The function manuals contain detailed descriptions on general topics relating to the SIMATIC S7-1500 and ET 200MPsystems.

Examples:

- Function Manual Diagnostics
- Function Manual Communication
- Function Manual Motion Control
- Function Manual Web Server
- Function Manual Cycle and Response Times
- PROFINET Function Manual
- PROFIBUS Function Manual

Product Information

Changes and supplements to the manuals are documented in a Product Information. The Product Information takes precedence over the device and system manuals.

You can find the latest Product Information on the S7-1500 and ET 200MP systems on the Internet (https://support.industry.siemens.com/cs/de/en/view/68052815).

Manual Collection S7-1500/ET 200MP

The Manual Collection contains the complete documentation on the SIMATIC S7-1500 automation system and the ET 200MP distributed I/O system gathered together in one file.

You can find the Manual Collection on the Internet. (https://support.industry.siemens.com/cs/ww/en/view/86140384)

SIMATIC S7-1500 comparison list for programming languages

The comparison list contains an overview of which instructions and functions you can use for which controller families.

You can find the comparison list on the Internet (https://support.industry.siemens.com/cs/ww/en/view/86630375).

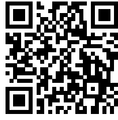
1.2 SIMATIC Technical Documentation

Additional SIMATIC documents will complete your information. You can find these documents and their use at the following links and QR codes.

The Industry Online Support gives you the option to get information on all topics. Application examples support you in solving your automation tasks.

Overview of the SIMATIC Technical Documentation

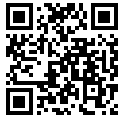
Here you will find an overview of the SIMATIC documentation available in SIOS:



Industry Online Support International (https://support.industry.siemens.com/cs/ww/en/view/109742705)

Watch this short video to find out where you can find the overview directly in SIOS and how to use SIOS on your mobile device:

Quick introduction to the technical documentation of automation products per video (https://support.industry.siemens.com/cs/us/en/view/109780491)



YouTube video: Siemens Automation Products - Technical Documentation at a Glance (https://youtu.be/TwLSxxRQQsA)

mySupport

With "mySupport" you can get the most out of your Industry Online Support.

Registration	You must register once to use the full functionality of "mySupport". After registration, you can create filters, favorites and tabs in your personal workspace.
Support requests	Your data is already filled out in support requests, and you can get an overview of your current requests at any time.
Documentation	In the Documentation area you can build your personal library.
Favorites	You can use the "Add to mySupport favorites" to flag especially interesting or frequently needed content. Under "Favorites", you will find a list of your flagged entries.
Recently viewed articles	The most recently viewed pages in mySupport are available under "Recently viewed articles".
CAx data	The CAx data area gives you access to the latest product data for your CAx or CAe system. You configure your own download package with a few clicks: • Product images, 2D dimension drawings, 3D models, internal circuit diagrams, EPLAN macro files • Manuals, characteristics, operating manuals, certificates • Product master data

You can find "mySupport" on the Internet. (https://support.industry.siemens.com/My/ww/en)

Application examples

The application examples support you with various tools and examples for solving your automation tasks. Solutions are shown in interplay with multiple components in the system - separated from the focus on individual products.

You can find the application examples on the Internet. (https://support.industry.siemens.com/cs/ww/en/ps/ae)

1.3 Tool support

The tools described below support you in all steps: from planning, over commissioning, all the way to analysis of your system.

TIA Selection Tool

The TIA Selection Tool tool supports you in the selection, configuration, and ordering of devices for Totally Integrated Automation (TIA).

As successor of the SIMATIC Selection Tools , the TIA Selection Tool assembles the already known configurators for automation technology into a single tool.

With the TIA Selection Tool , you can generate a complete order list from your product selection or product configuration.

You can find the TIA Selection Tool on the Internet. (https://support.industry.siemens.com/cs/ww/en/view/109767888)

SINETPLAN

SINETPLAN, the Siemens Network Planner, supports you in planning automation systems and networks based on PROFINET. The tool facilitates professional and predictive dimensioning of your PROFINET installation as early as in the planning stage. In addition, SINETPLAN supports you during network optimization and helps you to exploit network resources optimally and to plan reserves. This helps to prevent problems in commissioning or failures during productive operation even in advance of a planned operation. This increases the availability of the production plant and helps improve operational safety.

The advantages at a glance

- Network optimization thanks to port-specific calculation of the network load
- Increased production availability thanks to online scan and verification of existing systems
- Transparency before commissioning through importing and simulation of existing STEP 7 projects
- Efficiency through securing existing investments in the long term and the optimal use of resources

You can find SINETPLAN on the Internet (https://new.siemens.com/global/en/products/automation/industrial-communication/profinet/sinetplan.html).

See also

PRONETA Professional (https://support.industry.siemens.com/cs/ww/en/view/109781283)

2 Introduction

2.1 Operating instructions guide

Purpose of the documentation

These operating instructions supplement the manuals of the S7-1500 Software Controller and the CPU 1518(F)-4 PN/DP MFP with SIMATIC Industrial OS as of firmware version V2.9.

The information provided in these operating instructions enables you to:

- Install the Realtime Information Backbone (RIB) on a SIMATIC IPC with an S7-1500 Software Controller
- Use the Realtime Information Backbone (RIB) on an S7-1500 Software Controller or a CPU 1518(F)-4 PN/DP MFP
- Use libraries for C++ applications
- Use the 5 different modes of SFC65490 to connect RIB to the CPU user program

Basic knowledge required

The following knowledge is required to understand this documentation:

- General knowledge of automation technology
- Knowledge of the SIMATIC industrial automation system
- Knowledge of working with STEP 7
- Knowledge of working with Linux
- Knowledge of programming with C/C++

Validity of the documentation

This documentation is valid for users of the Realtime Information Backbone (RIB) and supplements the manuals of the following products:

CPU		Article number	RIB pre-installed
Software Controllers	1505SP	6ES7672-5DC12-0YA0	--
	1505SP F	6ES7672-5SC12-0YA0	--
	1507S	6ES7672-7AD02-0YG0	--
	1507S F	6ES7672-7FD02-0YG0	--
	1508S	6ES7672-8AD02-0YG0	--
	1508S F	6ES7672-8FD02-0YG0	--
Hardware CPUs	1518-4 PN/DP MFP	6ES7518-4AX00-1AB0	✓ (as of firmware version V2.9)
	1518F-4 PN/DP MFP	6ES7518-4FX00-1AB0	✓ (as of firmware version V2.9)

Note

Naming convention

Whenever the generic term "CPU" is used in this manual, we refer to both the Software Controllers and Hardware CPUs.

Definitions

We will use the following terms and definitions when explaining the RIB functional principle and example projects:

Name	Definition
Symbol	The smallest meaningful data element in the RIB environment for exchanging data A 'Symbol' is a shared variable between provider and consumer. The variable is described by a symbolic name, data type, data size and storage location in a shared memory.
Provider	An application connected to RIB that provides symbols
Consumer	An application connected to RIB that reads symbols from providers

3 Safety information

3.1 Security information

Siemens provides products and solutions with industrial security functions that support the secure operation of plants, systems, machines and networks.

In order to protect plants, systems, machines and networks against cyber threats, it is necessary to implement – and continuously maintain – a holistic, state-of-the-art industrial security concept. Siemens’ products and solutions constitute one element of such a concept.

Customers are responsible for preventing unauthorized access to their plants, systems, machines and networks. Such systems, machines and components should only be connected to an enterprise network or the internet if and to the extent such a connection is necessary and only when appropriate security measures (e.g. firewalls and/or network segmentation) are in place.

For additional information on industrial security measures that may be implemented, please visit (https://www.siemens.com/industrialsecurity).

Siemens' products and solutions undergo continuous development to make them more secure. Siemens strongly recommends that product updates are applied as soon as they are available and that the latest product versions are used. Use of product versions that are no longer supported, and failure to apply the latest updates may increase customers' exposure to cyber threats.

To stay informed about product updates, subscribe to the Siemens Industrial Security RSS Feed visit (https://www.siemens.com/cert).

3.2 Open-source software

Open-source software is used in the firmware of the product described. Open-source software is provided free of charge. We are liable for the product described, including the open-source software contained in it, pursuant to the conditions applicable to the product. Siemens accepts no liability for the use of the open-source software over and above the intended program sequence, or for any faults caused by modifications to the software.

For legal reasons, we are obliged to publish the original text of the license conditions and copyright notices. Readme_OSS files containing all relevant information on the open-source software used in this product are provided in the compressed folder "documentation.tar.gz".

Product overview / Software description 4

4.1 Integration of RIB

The Realtime Information Backbone (RIB) is an environment to exchange data between the CPU and realtime applications running on SIMATIC Industrial OS (IndOS) in parallel on the same device.

The creation of client applications is not restricted to a specific programming language. The programming language must support the exchange of JSON formatted strings over a TCP socket connection. Additionally, access to POSIX (Portable Operating System Interface) or interaction with a driver (VMM shared memory driver) must be possible. The communication protocol and data exchange concept are easy to implement.

Note

POSIX (Portable Operating System Interface)

If you do not install the Software Controller and hypervisor, only the Linux-based POSIX is available.

Example of data exchange between an IndOS application and a CPU

The following image shows how RIB is integrated into the system architecture. The example uses an IPC with IndOS and a CPU as RIB client.

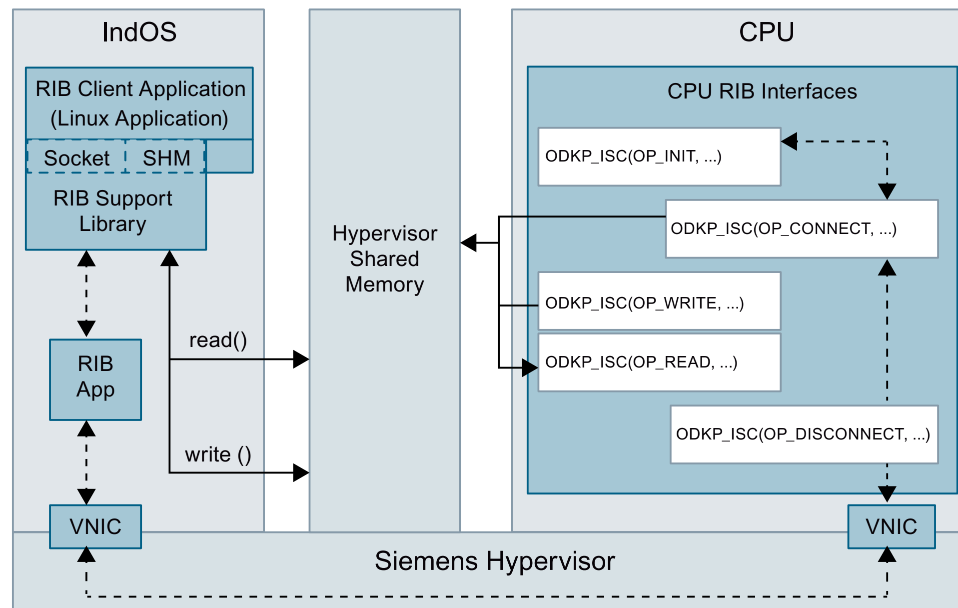


Figure 4-1 Overview of system architecture

Communication and the exchange of data between the RIB_App and the RIB client applications are facilitated by the use of the RIB Support Library. The RIB Support Library is installed together with the RIB_App.

For more information about the RIB Support Library, refer to chapter RIB Support Library (Page 47).

NOTICE
Root rights For the Software Controller, the RIB_App and the RIB Support Library do not require root rights by default. Therefore, other applications do not require root rights, either. For the CPU 1518(F)-4 PN/DP MFP, the RIB_App and the RIB Support Library do require root rights. The root user may modify permissions and ownerships in a way that the RIB_App and the RIB Support Library are also accessible to users without root rights. However, if additional actions (for example, setting process priorities) require root rights, the application developer and user must be aware that applications with root access can cause significant damage to the system.

System requirements

To use RIB, your system must meet the following requirements:

Category	Requirement
Operating system	SIMATIC Industrial OS ≥ V2.4
Supported CPUs	CPU 1505SP (F), CPU 1507S (F), CPU 1508S (F) as of V21.9 and CPU 1518(F)-4 PN/DP MFP as of V2.9.3
CPU firmware	as of V2.9
Supported SIMATIC Hardware	IPC 227G, IPC 427E, IPC 477E, IPC 477E Pro and BX-39A with at least 4 GB free memory space on the target storage medium and CPU 1515SP PC2 Open Controller
Supported TIA Portal versions	as of V17

SFC modes

The CPU provides the SFC ODKP_ISC to exchange data between the user program and Linux applications via the RIB.

This instruction provides methods to:

- Initialize ("OP_INIT") and establish ("OP_CONNECT") a connection to RIB
- Define symbols for data exchange using data blocks
- Read ("OP_READ") and write ("OP_WRITE") data of symbols
- Disconnect "OP_DISCONNECT" from RIB

4.2 Realtime data exchange

4.2.1 Realtime data exchange concept

The RIB and one or more RIB client applications exchange their data in a shared memory area. This area is called the lifetime buffer. The following image shows the functional principle of the lifetime buffer:

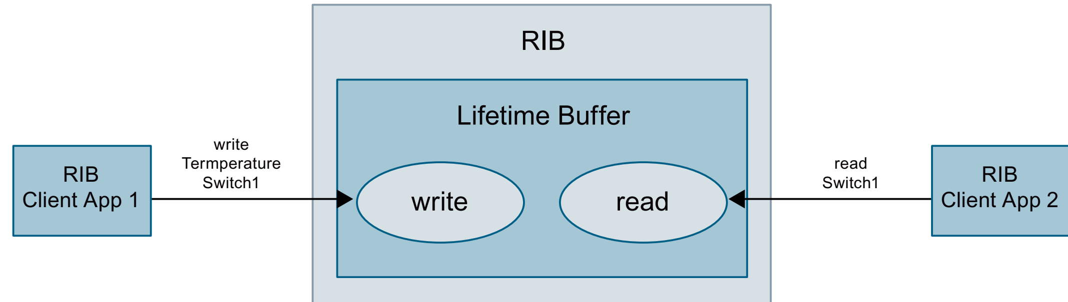


Figure 4-2 Realtime data exchange

The lifetime is the time span during which a provider must not overwrite a buffer element of the lifetime buffer. The buffer element is valid for at least this time span. During this time span, the data within the buffer element is consistent.

Note

Shared memories

Shared memories used for data exchange are not exclusively available for the applications that opened it or access it as a consumer via RIB. Therefore, malicious software might be able to modify content of an existing lifetime buffer unnoticed by the user. If you exchange sensitive data via the RIB, you must take appropriate measures to secure the data.

Note

Deterministic data exchange

The RIB_App and the RIB Support Library do not guarantee deterministic data exchange in cases where system utilization is high or very large data blocks are exchanged. The timing constraints are highly dependent on system utilization and process priorities. Therefore, the user must make sure that realtime requirements are met. Furthermore, the user must also consider other circumstances, for example interrupts from the Siemens Hypervisor, when configuring the system.

4.2.2 Adjustments for correct realtime behavior

For realtime applications to work properly, a Linux system with the "RT-Preempt" patch installed must be configured accordingly. To enable correct realtime behavior, the following adjustments apply:

Core pinning

Linux systems running with the SIMATIC RT-VMM (Hypervisor) have some particularities that must be taken into account:

- Linux systems generally use all but one physical available processor cores. If there is, for example, a processor with 4 physical cores without hyper-threading, Linux will use 3 cores. With hyper-threading, there would be six logical cores available.
- The Linux OS shares the first physical core with the hypervisor (core 0 if no hyper-threading is available and core 0 and 1 with hyper-threading).
- The hypervisor has priority over the Linux OS.
- The Software Controller runs on a separate processor core and is not interrupted by the hypervisor.

Due to these particularities, it is advisable to pin the execution of the RIB_App to core 0 because the RIB_App does not require realtime performance. To do this, you can start the RIB_App as follows:

```
taskset -c 0 RIB_App
```

For the performance of realtime applications not to be affected by the hypervisor, these applications should be pinned to a core that is not shared with the hypervisor (for example, core 1 or 2 when using a 4-core processor without hyper-threading). To do this, proceed as follows:

```
taskset -c 1 RT_Application
```

Note

CPU 1518(F)-4 PN/DP MFP

The CPU 1518(F)-4 PN/DP MFP has one CPU core available for Linux. Therefore, core pinning is not useful for this platform.

Setting realtime priority

Non-root users are not allowed to use RT priorities. To change this for user "pt1", add the following line to the file */etc/security/limits.conf*:

```
pt1 - rtprio 30
```

30 is the maximum priority that the user "pt1" can set (100 is the maximum priority and 0 is the default priority).

The priority on which a realtime application should run can be set in its code. The following lines should be added to run at the beginning of the CPU user program:

//Set process priority

```
struct sched_param param;
```

```
param.sched_priority = 30;
sched_setscheduler(0, SCHED_FIFO, &param);
```

Memory locking

To lock part or all of the virtual address space of the calling process into RAM, which prevents memory from being paged to the swap area, add the code `pt1 - memlock -1` to the file: */etc/security/limits.conf*:

"-1" means that the user "pt1" can lock an "unlimited" amount of space into RAM.

For an application to lock all of its virtual address space into RAM, add the following code to run at the beginning of the application:

//Lock process memory to not be stored on disc

```
mlockall(MCL_CURRENT | MCL_FUTURE);
```

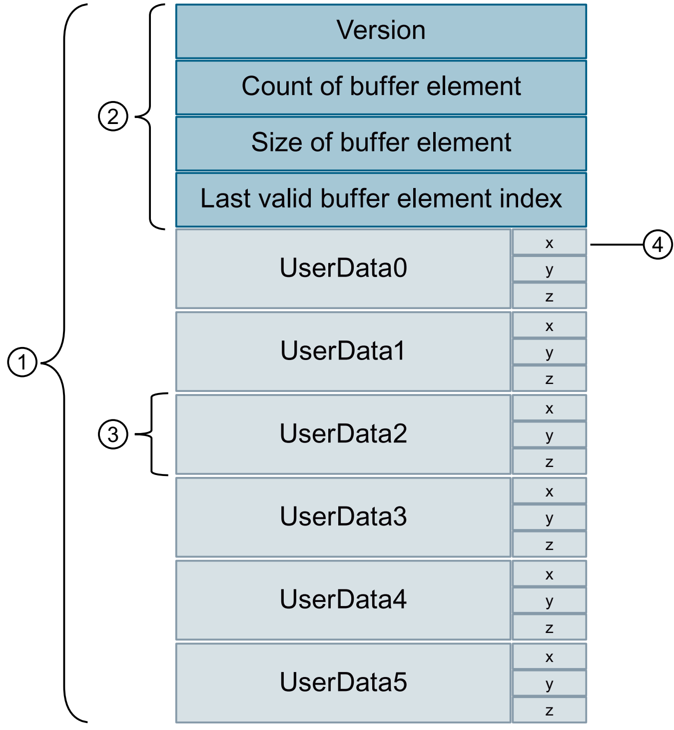
Note

Locking insufficient memory space

If you try to lock less memory space than you consume, the application will not work anymore or might crash.

4.2.3 Lifetime buffer

The lifetime buffer stores the data to be shared. The following image shows the components of the lifetime buffer.



① Lifetime buffer

② Header

③ Buffer element

④ Symbol

Figure 4-3 Buffer layout

Header

The header contains the following information:

Element	Size in bytes	Description
Buffer version	2	Contains the version of buffer implementation, for example 1.0 The first byte represents the major part of the version (1). The second byte represents the minor part of the version (0). Every consumer that opens this lifetime buffer for reading must have the same major version number. If not, the generation of the `Reader` object in the `RibClient` fails with the error `InvalidBufferVersion`.
Buffer type	2	In addition to the buffer version, the header of each lifetime buffer contains a buffer type. The first version of the RIB Support Library only supports the lifetime buffer. This is set by provider applications by default when using the RIB Support Library. If a consumer tries to open a lifetime buffer with a different buffer type, the generation of the `Reader` object in the `RibClient` will fail with the error `InvalidBufferType`.
CountOfBufferElements	4	Defines number of buffer elements
SizeOfABufferElement	4	Defines size of each buffer element in bytes
LastValidBufferElementIndex	4	Defines latest updated buffer element

Buffer elements

A buffer element contains all symbols that the RIB client application provides. The symbols must have the structure as described in chapter Connecting client applications (Page 35).

A buffer element, e.g. UserData0, is a simple array of bytes in which individual symbols are stored in an array with no gaps. Symbols whose types require more than 1 byte are stored in standard Intel format. When using the RIB Support Library, the system arranges the individual symbols as a byte array automatically. For read and write access, the offset and the data type are relevant. When using the RIB Support Library, the pointers for reading and writing data are provided on the basis of the offset and the data type.

Calculating lifetime buffer size

The maximum size of VMM shared memory is 8 MB.

Use the following formula to calculate the size of the lifetime buffer:

Size-of-Lifetimebuffer = Size-of-Header + Number-of-Segments * Segment-size

where:

Size-of-Header = 16 bytes

Number-of-Segments = 3 + Lifetime / Cycle-time

Segment-size = Sum-of-all-symbol-sizes (rounded up to the next multiple of 8)

Compiler settings

The compiler settings must be the same throughout all applications used. The recommended compiler settings are stored in the file *CMakeLists.txt* of the example applications.

4.2.4 Writing data to lifetime buffer

The providing application writes its data into a buffer element. After writing the data to the buffer element, the header field "LastValidBufferElementIndex" must be updated. Now the lifetime starts to be counted. The buffer element is allowed to be written again after the lifetime has expired. The writing algorithm must ensure that a buffer element does not change within the lifetime.

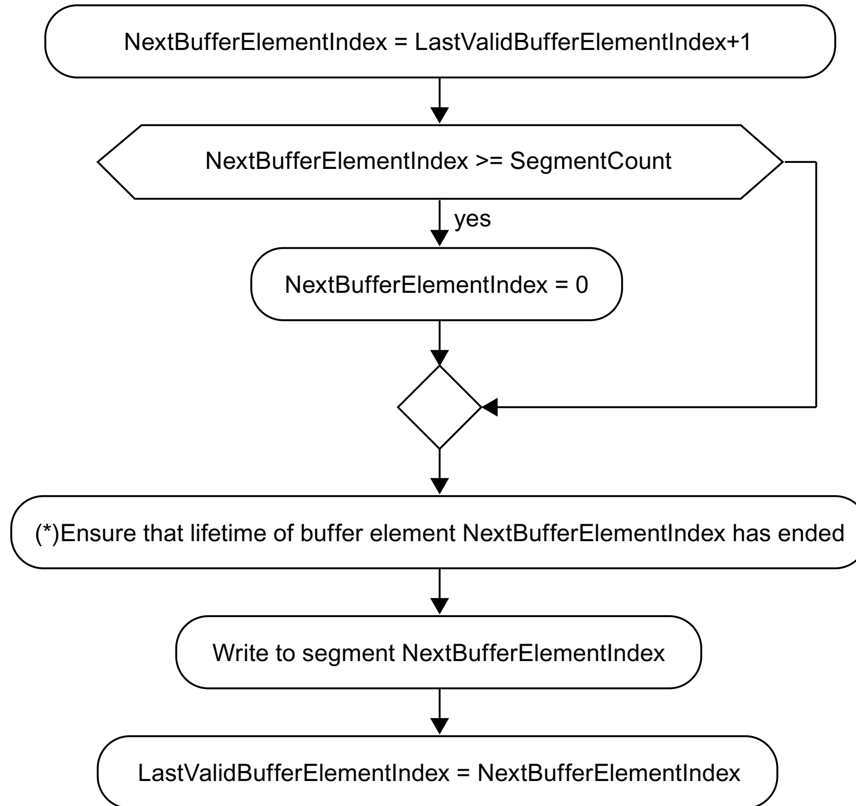
Note

Lifetime limits

We recommend not to exhaust the lifetime to its limits. Even though the algorithm still works, other topics like clock accuracy, the processor's memory management and other system hardware effects might compromise the user's algorithm.

Example algorithm

The following flow chart shows a possible implementation of the write algorithm. This algorithm describes a ring buffer.



4.2.5 Reading data from lifetime buffer

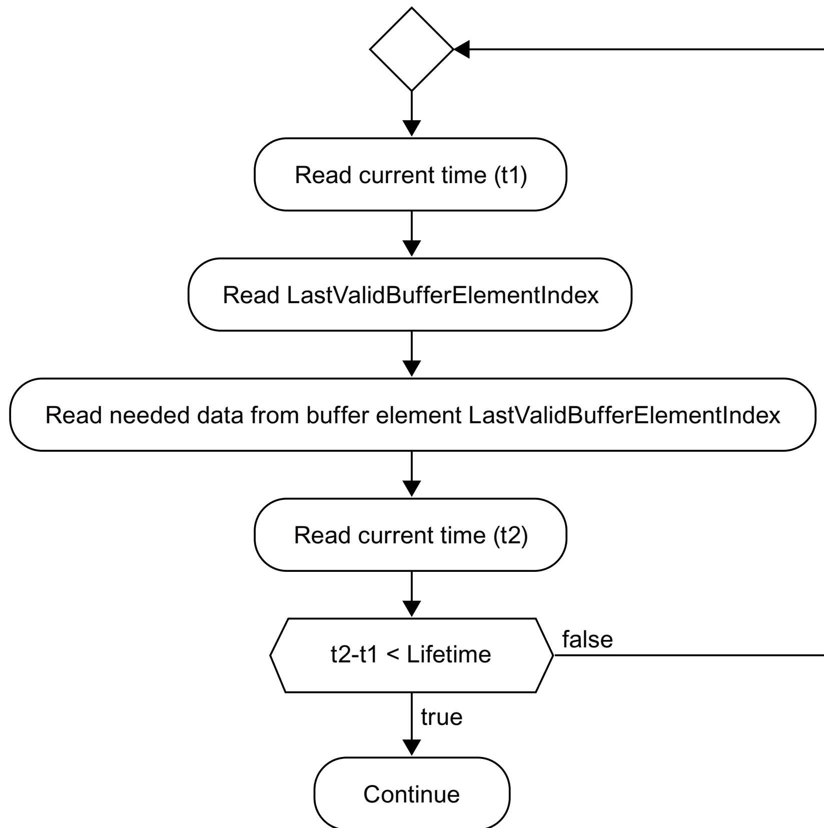
When reading data from the lifetime buffer, the following two actions must be carried out within the specified lifetime:

1. Reading "LastValidBufferElementIndex" from the header of the lifetime buffer.
2. Reading the buffer element indexed by "LastValidBufferElementIndex".

If these operations are finished within the lifetime, the read data is consistent. Otherwise, the reading process must be repeated.

Example algorithm

The following flow chart shows an example reading process that a consumer must follow:



4.2.6 Buffer example

The following image shows an example where the provider application writes data to and the consumer reads data from the lifetime buffer. The guaranteed buffer element lifetime is 3 ms. The cycle time of the provider is 1 ms. The number of buffer elements is based on the formula shown in section Lifetime buffer (Page 18):

Number-of-Segments = 3 + Lifetime / Cycle-time

Number-of-Segments = 3 + 3 / 1

For this reason, 6 buffer elements are provided.

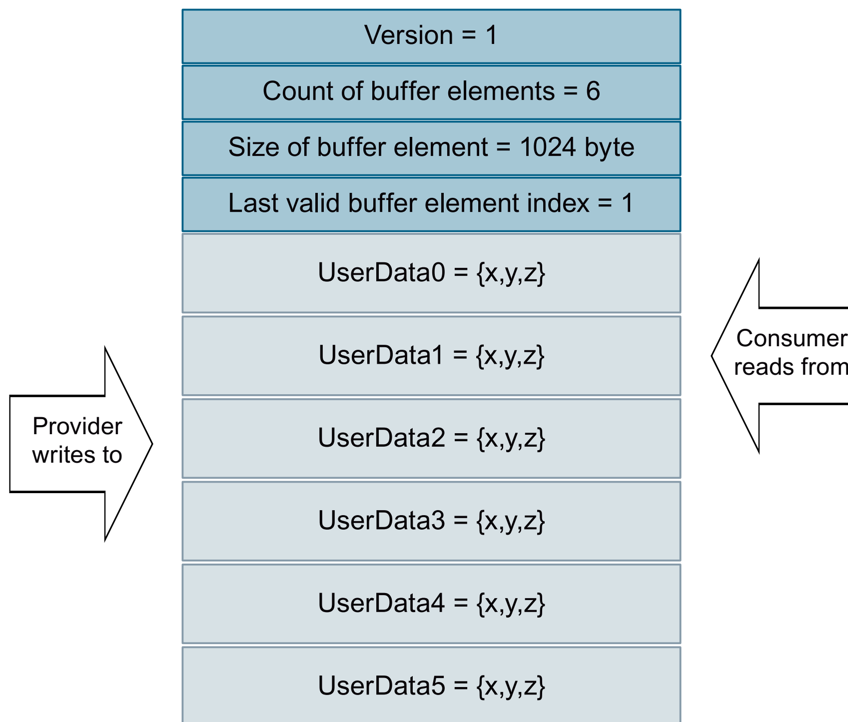


Figure 4-4 Writing and reading data from buffer

5 Software installation and uninstallation

5.1 Installing RIB on CPU 1518(F)-4 PN/DP MFP

On a CPU 1518(F)-4 PN/DP MFP, the RIB_App and RIB Support Library are already pre-installed and require root rights.

The files under the path mentioned in the table below require root rights to be changed. If you want to allow other users to change these files, add those users and modify the access rights of the preinstalled files by calling the commands mentioned in the table below as root:

Path	Command for modifying access rights
/usr/bin/RIB_App	chmod 755 /usr/bin/RIB_App
/usr/lib/librib_support.so.1.0, incl. symlink	chmod 755 /usr/lib/librib_support.so.1.0
/usr/lib/libsimatic-vmm-shmem.so.1	chmod 755 /usr/lib/libsimatic-vmm-shmem.so.1
/usr/bin/s7_RIB_memory_cleaner	chmod 755 /usr/bin/s7_RIB_memory_cleaner

Note

Creating dedicated user groups

You can alternatively assign ownership and access rights to files to a dedicated user group.

5.2 Installing RIB on Software Controller

The following chapter describes how to install the RIB_App and the RIB Support Library on an S7-1500 Software Controller 1505SP (F), 1507S (F) or 1508S (F).

Installing RIB_App and RIB Support Library

When installing the S7-1500 Software Controller, the RIB setup is copied to <mount_point>/SWCPU/RIB.

For installing the RIB_App and RIB Support Library, use the binary "RIBSetup".

Note

Root privileges

The binary must be executed with root privileges.

```
sudo ./RIBSetup
```

Result after successful installation

After successful installation, the following components are copied to the following paths:

Component	Path
RIB_App	/usr/bin/RIB_App
C++ support library [1)]	/usr/lib/librib_support.so.[version], including a link /usr/lib/librib_support.so
Simatic VMM shared memory library	/usr/lib/libsimatic-vmm-shmem.so.1
Template applications [2)]	/home[/user]/rib
Copyright file	/usr/share/doc/ribenvironment/copyright
Changelog	/usr/share/doc/ribenvironment/changelog.gz
VMM shared memory access rules	/etc/udev/rules.d/99-simatic-shm.rules
s7_RIB_memory_cleaner	/usr/bin/s7_RIB_memory_cleaner

1) It is allowed to link the rib_support.so. files dynamically, but not statically

2) This folder contains application examples for the customer. If there is already a folder for templates, it will not be overwritten by the installation. The templates are copied to a new folder with the same name. The name is then followed by an underscore and the first integer that is not used, starting with "1", e.g. /home[/user]/rib_1.

User group 'rib_operators'

During the first installation of the RIB_App, the user group 'rib_operators' is created. The user who executes the installation is added automatically to the group 'rib_operators'. The administrator of the system is allowed to add and remove users to this group.

VMM shared memory driver

Additionally, a package for the VMM shared memory driver is installed and the driver (simatic-vmm-shmem) is loaded.

- If `dkms` is installed, the appropriate VMM shared memory driver is installed automatically.
- If `dkms` is not installed, the version of the VMM shared memory driver must strictly match the current kernel version. If the kernel is upgraded by the user, a reinstallation of the RIB with the correct VMM shared memory driver becomes additionally necessary.

Access to the 'Simatic-VMM shared memory driver' is restricted. Only users who belong to the 'rib_operators' group are allowed to create or access 'Simatic-VMM shared memory'.

Access control for the Hypervisor shared memory is realized by the tool 'udev' and the configuration file located in: '/etc/udev/rules.d/99-simatic-shm.rules'. This file contains the following configuration:

```
bash
KERNEL=="simatic-vmm/*", SUBSYSTEM=="simatic-vmm/*",
GROUP=="rib_operators", MODE="0660"
```

The file '/etc/udev/rules.d/99-simatic-shm.rules' is always created from scratch during the installation process with the default settings mentioned above. If you want to set your own access rules for the 'Simatic-VMM shared memory driver', you must define your own rule file that is not touched by the installation or uninstallation of the RIB_App.

If loading the driver for the VMM shared memory fails, you can only use RIB with Linux applications, but not with the Software Controller. The following error message is printed to the logfile and console:

```
Simatic-VMM shared memory driver installation FAILED!
```

```
Communication to Simatic Software Controller is not possible via RIB
However RIB can be used on Linux only systems without restriction
```

Installation logs

The installation logs all important information to the console and to the log file. You can find the log file under the following path:
`/home[/user]/.siemens/rib_installation.txt`

5.3 Compatibility rules

Versioning schema

The RIB_App and RIB Support Library use the version schema 'V.x.y.z'. The meaning of the version letters is as follows:

Version letter	Meaning
'V'	Prefix that is kept through all versions
'x'	Represents the major version (starts with 0 and will only be changed if incompatible API changes are made)
'y'	Represents the minor version (is increased if significant functionality is added in a backwards compatible manner)
'z'	Represents the patch level (is changed, for example, when doing backwards compatible bug fixes)

Version compatibility rules

In general, the RIB_App and RIB Support Library will be released together and should have the same version. However, 'RIB_App V1.y.z' and 'RIB Support Library V1.y.z' might face newer versions like 'RIB_App V2.y.z' and 'RIB Support Library V2.y.z' in the future.

The following matrix summarizes the version compatibility:

	RIB Support Library V1.y.z	RIB Support Library V2.y.z	Software Controller V21.9	Software Controller V30.0
RIB_App V1.y.z	*compatible*	*incompatible*	*compatible*	*compatible*
RIB_App V2.y.z	*compatible*	*compatible*	*compatible*	*compatible*

Compatibility handling for JSON messages

The version information is a mandatory attribute that must be available in the top level of each JSON string that is sent by a client application to the RIB_App.

The RIB_App evaluates each received JSON string which contains a version information. If the version is available and evaluated as compatible, the message is accepted and processed accordingly. Otherwise, the following two error cases exist:

- The version is regarded as incompatible
- No version attribute can be found

In these cases, the RIB_App will respond with a general response message that is specified as follows:

Key	Value	Description
"Type"	"GeneralResponse"	Message type; for a 'general response', type must be `GeneralResponse`
"Version"	"1.0"	Protocol version
"RIBInformation"	"RibInformationData"	Contains all information regarding connection to RIB

'RibInformationData' contains the following information:

Key	Value	Description
"RIBPid"	"1111"	Process identification number of RIB_App in Linux operation system
"RIBVersion"	"1.0"	Version of RIB_App, for example `1.0`
"Result"	"error"	Indicates that the request is invalid
"ErrorMessage"	"Detailed error message"	Provides more details about the error

The "DetailedErrorMessage" contains additional information about the error:

- If no "Version" was found, "DetailedErrorMessage" contains the following string: "Version not available"
- If "Version" is regarded as incompatible, the "DetailedErrorMessage" contains the following string: "VersionNotSupported 'GivenVersion'"
  ('GivenVersion' is the version sent by the client application)

5.4 Uninstalling RIB from Software Controller

Uninstalling RIB_App and RIB Support Library

For uninstalling the RIB_App and RIB Support Library, execute "RIBSetup" with the parameter '-- -u', for example:

```
sudo ./RIBSetup -- -u
```

Successful uninstallation removes the following components from the system:

Component	Path
RIB_App	/usr/bin/RIB_App
C++ support library	/usr/lib/librib_support.so.[version], including a link /usr/lib/librib_support.so
Simatic VMM shared memory library	/usr/lib/libsimatic-vmm-shmem.so.1
Copyright file	/usr/share/doc/ribenvironment/copyright

Component	Path
Changelog	/usr/share/doc/ribenvironment/changelog.gz
VMM Shared Memory Access Rules	/etc/udev/rules.d/99-simatic-shm.rules

Note

Remaining components after uninstallation

The component "Template applications" will not be removed by the uninstallation.

The user group 'rib_operators' will not be automatically removed from the system, either. This way, the list 'rib_operator' is still available when reinstalling or updating RIB. If you want to remove the list, the list must be removed manually by the administrator.

Uninstallation while RIB_App is running

If there is a RIB_App running during uninstallation, the system asks you if the running instance should be terminated before by the uninstallation. Confirm this prompt with 'y' or 'Y' to continue and 'n' or 'N' to abort uninstallation. You can also skip the confirmation and continue the uninstallation process anyway by adding the parameter 'y':

```
sudo ./RIBSetup -- -u -y
```

VMM shared memory driver

The package for the VMM shared memory driver is also uninstalled and the driver (simatic-vmm-shmem) is unloaded. If uninstallation of the VMM shared memory fails, the following message will be printed to the logfile and console:

```
Simatic-VMM driver uninstallation FAILED!

Please ignore message above, if you want to work on a system
without Simatic VMM.
```

Note

Uninstallation of the Software Controller

Note that an uninstallation of the S7-Software Controller will also remove the RIB installer. If you want to uninstall the Software Controller, we recommend that you uninstall the RIB first.

Uninstallation logs

The uninstallation logs all important information to the console and to the log file. You can find the log file under the following path:

```
/home[/user]/.siemens/rib_installation.txt
```

RIB_Application 6

6.1 RIB_Application

The RIB_Application (RIB_App) is an application that runs on Linux operating systems and is the basic environment for all RIB clients.

The RIB_App brings different RIB client applications together in order for them to exchange data. RIB client applications are not restricted to a one-to-one relationship. RIB client applications can have many-to-many relationships, as well.

Starting RIB_App

For starting the RIB_App, run the "RIB_App" executable on the Linux operating system. Starting the RIB_App is possible without root privileges. However, you can only start one RIB_App instance at a time.

Note

Starting RIB_App on CPU 1518(F)-4 PN/DP MFP

For the CPU 1518(F)-4 PN/DP MFP, root rights are required to start RIB_App. If you want to allow non-root users to start RIB_App, change the access rights of the following files to 755:

- /usr/bin/RIB_App
- /usr/lib/librib_support.so.1.0
- /usr/lib/libsimatic-vmm-shmem.so.1

TCP communication

The port (port number 27567) that the RIB application opens for communication with client applications is currently available for all network interfaces that can be accessed by the RIB.

Furthermore, the port of the RIB_App is currently not adjustable. If the port 27567 is already being used, the RIB_App will abort with an error.

Note

Allow TCP communication for the Software Controller

Note that you must explicitly allow the Software Controller to establish a connection to the RIB_App.

To allow TCP communication, proceed as follows:

1. Run "sudo ufw status" to show the firewall status.
2. Run "sudo ufw allow from <SWCPU_VNIC_IP_ADDRESS> to <LINUX_VNIC_IP_ADDRESS> port 27567 proto tcp" to add a rule to allow TCP communication over port 27567.

The information above is also printed out at the end of successful installation.

The CPU 1518(F)-4 PN/DP MFP already includes these firewall rules.

Terminating RIB_App

To terminate the RIB_App, you can use, for example, the key combination CTRL+C to send a "SIGINT" to the Linux process, or alternatively send a "SIGTERM" signal.

The RibClient notices if the connection to the RIB_App is terminated. A termination of the connection can happen, e.g. if the network interface is disabled after activation of the RibClient or if the RIB_App itself is terminated. In this case, a log message is printed to the console. However, read and write access to the Lifetime Buffers is still possible.

The RibClient will not automatically reconnect to the RIB_App afterwards.

RIB_App startup parameters

On startup, you can configure different parameters. These parameters specify the behavior of the RIB_App. The following table lists all supported parameters:

Parameter	Shortcut	Description
`--help`	`-h`	Displays the RIB_App help in the console
`--verbose`	`-v`	Displays additional debugging logs for the RIB_App in the console
`--version`	`-V`	Displays the RIB_App version in the console
`--libraryversion`	`-lv`	Displays the version of the current RIB support library used by the RIB_App in the console
`--lifetime`	`-l`	Starts the RIB_App and defines the maximum guaranteed lifetime in milliseconds of the buffer elements in the lifetime buffer for the RIB environment. If you do not use this parameter, the default value is 10 milliseconds. The allowed minimum is 1 millisecond. The maximum lifetime is not restricted. A syntax example of a lifetime of 20 milliseconds is: `RIB_App -l 20`
`--waittime`	`-w`	Starts the RIB_App and defines the timeout waiting time for pending socket responses in the RIB_App. If you do not use this parameter, the default value is 15 seconds. The allowed minimum is 1 second. The maximum timeout waiting time is not restricted.

Restrictions

The RIB_App has the following restrictions:

- Accessing the RIB_App has only been tested with the IPv4 protocol.
- The total number of client applications allowed to establish a socket connection with the RIB_App is not restricted. The guaranteed number of simultaneous applications that create a new socket connection to the RIB_App is 128 clients.

6.2 Diagnostic data

The RIB_App offers access to diagnostic data. You can use this data for the development of diagnostic applications. The RIB_App returns a list. This list describes all connected applications with their provided and requested symbols.

Note

Note that the following requests are only a description of features that the RIB_App offers on the low-level API. It is the task of the user to implements these requests, as they are not provided by the RIB Support Library.

Diagnostic data request

The RIB_App accepts a diagnostic data request of client applications using an established TCP socket connection. The diagnostic data request must be formatted as a JSON string and must contain the following information:

Key	Value	Description
"Type"	"DiagnosticDataRequest"	Type of message; must be `DiagnosticDataRequest`
"Version"	"1.0"	Protocol version

Diagnostic data response

The RIB_App responds to a diagnostic data request with a diagnostic data response. The provided and requested symbols are structured as in "ConnectToRIBConfig". The RIB_App sends the disconnect response message as a JSON formatted string. The string contains the following information:

Key	Value	Description
"Type"	"DiagnosticDataResponse"	Type of message; must be `DiagnosticDataResponse`
"Version"	"1.0"	Protocol version
"RIBInformation"	"RibInformationData"	Contains all information regarding connection to RIB
"DiagnosticData"	"DiagnosticData"	Contains all available diagnostic data of the applications currently connected to RIB

"RibInformationData"

The value "RibInformationData" contains the following information:

Key	Value	Description
"RIBPid"	"1111"	Process identification number of RIB_App in Linux operation system
"RIBVersion"	"1.0"	Version of RIB_App, for example `1.0`
"Result"	"DiagnosticResponse"	Indicates that a diagnostic data response is provided
"ErrorMessage"	"Detailed error message"	Detailed error message in case of an error (optional)

"DiagnosticData"

The value "DiagnosticData" contains the following information:

Key	Value	Description
"Type"	"ConnectToRIBConfig"	Diagnostic data of all applications connected to RIB
"Version"	"1.0"	Protocol version
"<ApplicationName>"	"ApplicationDescription"	Each application connected to RIB contains a complete application description including all provided and requested symbols of client application

6.3 General response

General response message

When the RIB_App receives a client message with the following characteristics, the RIB_App responds with a 'general response' message indicating that the received string cannot be processed:

- The JSON format of the received string is invalid
- The message type is unknown
- The type attribute is completely missing

The RIB_App sends the 'general response' message as a JSON formatted string containing the following information:

Key	Value	Description
"Type"	"GeneralResponse"	Type of message, for a 'general response', type must be `GeneralResponse`
"Version"	"1.0"	Protocol version
"RIBInformation"	"RibInformationData"	Contains all information regarding connection to RIB

'RibInformationData' contains the following information:

Key	Value	Description
"RIBPid"	"1111"	Process identification number of RIB_App in Linux operation system
"RIBVersion"	"1.0"	Version of RIB_App, for example `1.0`

Key	Value	Description
"Result"	"GeneralError"	Indicates that an error occurred after interpreting the string sent from the client App to the RIB_App
"ErrorMessage"	"Detailed error message"	InvalidMessageType, InvalidJsonString, TooLongMessage, VersionNotSupported or AttributeMissing

6.4 Global RIB environment

Within the global RIB environment, the RIB_App has the following functions:

- Giving client applications access to the RIB environment settings
- Hosting and distributing global RIB environment parameters
- Accepting requests for global RIB environment configuration data in the form of a JSON formatted string
  The string must contain the following parameters:

Key	Value	Description
"Type"	"ConfigDataRequest"	Requests RIB ConfigData
"Version"	"1.0"	Protocol version

The RIB_App creates and sends a response string that contains the global RIB environment. This message is a JSON formatted string and contains the following parameters:

Key	Value	Description
"Type"	"ConfigDataResponse"	Responds to RIB ConfigData
"Version"	"1.0"	Protocol version
"ConfigData"	"RibConfigData"	Contains all global RIB configuration data

For RIB version V1.0, the "RibConfigData" in the response message contains the following parameter:

Key	Value	Description
"BufferElementLifeTime"	"10"	The maximum guaranteed lifetime in milliseconds of the buffer elements in the shared memory. It is defined for all applications in the current RIB environment. The default value is `10` (10 ms). The value is distributed as an unsigned integer. Only positive values are possible. You can only set this value when starting the RIB_App.

6.5 Connecting client applications

For connecting to the RIB environment, the client application must send a connection request via an established socket connection. The RIB_App then processes this request and returns a connection response to the client application.

Requirements

To be able to connect to the RIB_App, the client application must meet the following requirements:

- The connection request of the client application uses an established TCP socket connection
- The connection request contains an application configuration formatted as a JSON string with a maximum length of 1 MiB
  If the size of this string is exceeded, the RIB_App logs the following error message: "Received string data too long or missing zero termination."
  Afterwards, the RIB_App closes the socket connection to the client application.

JSON string

The JSON string contains the following information:

Key	Value	Description
"Type"	"ConnectToRIBConfig"	Type of message; must be `ConnectToRIBConfig` for connection request
"Version"	"1.0"	Protocol version
"<ApplicationName>"	"ApplicationDescription"	All configuration data needed to register an application to RIB; It is not mandatory that the name matches the process name.

"ApplicationDescription"

The value "ApplicationDescription" contains the following information:

Key	Value	Description
"Type"	"ApplicationData"	Application data type is described here
"PID"	"<1234>"	Process identification number of app in Linux operating system
"Description"	"some text"	Optional: App can be described here
"Version"	"4.5"	Optional: App version
"Manufacturer"	"CompanyName"	Optional: Name of application creator, for example company name
"Provides"	"ProvidedData"	Optional: Describes the data provided by the app
"Requests"	"RequestedData"	Optional: Describes the data requested by the app

"ProvidedData"

The value "ProvidedData" contains at least one identifier of a shared memory and a corresponding shared memory description:

Key	Value	Description
"<IdentifierOfTheSharedMem>"	"SharedMemoryDescription"	Describes shared memory containing provided data

"SharedMemoryDescription"

The value "SharedMemoryDescription" contains the following information:

Key	Value	Description
"Type"	"Provide"	The provided type is described here
"Signal"	"<SignalId>"	Signal ID; not used yet; -1
"CycleTimeInMicroseconds"	"<CycleTime>"	Optional: Cycle time in microseconds used for writing data
"Description"	"some text"	Optional: Description of shared memory
"Version"	"1.0"	Optional: Version of provided data
"Symbols"	"SymbolList"	List of symbol descriptions, provided in shared memory

"SymbolList"

The value "SymbolList" contains a list of all symbols provided by the application in the shared memory to which it belongs. The symbol description of a provided symbol must contain the following information:

Key	Value	Description
"<SymbolName>"	"SymbolDescription"	Symbol name; must be unique in the RIB system

"SymbolDescription"

A "SymbolDescription" as part of a connection request contains the following data:

Key	Value	Description
"Offset"	"0"	Offset of symbol in shared memory in bytes
"Size"	"1"	Size of symbol in shared memory in bytes
"Type"	"uint8_t"	Type of symbol; in first version only simple value types are allowed

"Size" and "Type"

The "Size" attribute represents the complete size of the symbol in the shared memory. The "Size" is calculated from the "Type" size in bytes.

Note

Array

In case of an array, the number of elements ("ArrayCount") in this array must be calculated as follows:

```
Size = ArrayCount * TypeSize
```

Supported data types for symbols

The following data types are allowed to be used for symbols:

Category	Data types	Size in bytes
Signed integer	int8_t, int16_t, int32_t, int64_t	1, 2, 4, 8
Unsigned integer	uint8_t, uint16_t, uint32_t, uint64_t	1, 2, 4, 8
Floating points	float, double	4, 8

You can also use an array of these supported data types.

Note

BOOL not supported

Note that the data type BOOL is not supported.

6.6 Data processing and symbol matching

After receiving a connection request that contains configuration data from a client application, the RIB_App processes the JSON string.

Process steps

When processing data and matching symbols, the RIB_App progresses through the following steps:

- Checks the incoming JSON based strings and messages
  If the message is not supported or contains an error, the RIB_App does not process the string. The RIB_App returns an error to the client application in the 'general response' message.
- Detects and stores all information from the client application regarding provided symbols and additionally needed data (for example application data, shared memory name etc.)
- Detects and remembers all information from the client application regarding requested symbols and additionally needed data (for example application data, etc.)
- Matches the requested symbols with the already known provided symbols based on the symbol name (case-sensitive)
- Matches newly provided symbols with already known requested symbols based on the symbol name (case-sensitive)

6.7 Connection response

Based on the result of the JSON message data processing, the RIB_App generates a response message and sends it back to the client application. The RIB_App sends at least one response for each connection request of a client application. The RIB_App response message is restricted to a maximum length of 1 MiB.

Note

String size

Note that the number of consumed symbols might be restricted by the 1 MiB length of the JSON string and the length of symbol names, descriptions texts, etc.

If the string size is exceeded, the RIB_App logs the following error message: *"The message to be sent is too long. Please reduce number of provided/requested symbols or the number of applications."* The following response message containing a 'General Error' is returned to the client application: *"TooLongMessage RIB_App response would be too long."*

The RIB_App sends the connection response message as a JSON formatted string. The string contains the following information:

Key	Value	Description
"Type"	"ConnectToRIBResult"	Type of message; must be `ConnectToRIBResult` for the connection response
"Version"	"1.0"	Protocol version
"RIBInformation"	"RibInformationData"	Contains all information regarding connection to RIB
"DataProviderAvailable"	"DataProviderAvailableData"	Contains all information regarding available data providers for previously requested symbols (optional)

"RibInformationData"

The value "RibInformationData" contains the following information:

Key	Value	Description
"RIBPid"	"1111"	Process identification number of RIB_App in Linux operating system
"RIBVersion"	"1.0"	Version of RIB_App, for example `1.0`
"Result"	"<Connected or Error>"	Indicates whether connection was successful (`Connected`) or not (`Error`)
"ErrorMessage"	"A detailed error message"	Optional: Detailed error message in case of error

"DataProviderAvailableData"

The value "DataProviderAvailableData" is optional. If data providers are available, this value is part of the message. If no data providers are available yet or no symbols were requested, this value is not part of the message.

The value "DataProviderAvailableData" contains the following information:

Key	Value	Description
"Symbols"	"SymbolList"	List of symbol descriptions

The "SymbolList" contains a set of symbol descriptions. The number of symbols in this list is not restricted. However, the maximum number should match the number of symbols requested by the client application.

Each symbol is described as a pair of the symbol name and a description of the symbol details:

Key	Value	Description
"<SymbolName>"	"SymbolDescription"	All details describing a symbol in the RIB environment

A "SymbolDescription" contains the following information:

Key	Value	Description
"Offset"	"<AddressOffset>"	Offset of symbol in shared memory in bytes; 32 bit unsigned integer value
"Size"	"<SizesInBytes>"	Size of symbol in shared memory in bytes; 32 bit unsigned integer value
"Type"	"<SymbolType>"	Type of symbol; in first version only simple value types and arrays of them are allowed
"ShmId"	"<SharedMemoryId>"	Name of shared memory where requested symbol is in

"Size" and "Type"

The "Size" attribute represents the complete size of the symbol in the shared memory.

Note

Array

In case of an array, the number of elements ('Count') in this array must be calculated from "Size" and "Type" in bytes.

"Size" and "Type" can be calculated the following way:

```
math

ArrayCount = Size / TypeSize
```

Registration results

Registering an application to RIB can produce the following results:

- The registration of the application to RIB is successful.
- The registration of the application to RIB is not successful.
  Possible error reasons are:
  - Error in JSON message, for example invalid content, malformed JSON, etc
  - Application with same name already registered to RIB
    Solution: If the application was started twice, set or check a different application name
  - Symbol with same name already registered to RIB
    Solution: Set a different symbol name.

6.8 Disconnecting client applications

Disconnect request

For disconnecting a client application, the client application sends a disconnect request to the RIB_App via the established socket connection. The RIB_App processes this message and cleans the internal data storage.

The disconnect request is formatted as a JSON string and contains the following information:

Key	Value	Description
"Type"	"DisconnectFromRIB"	Type of message; must be `DisconnectFromRIB` for disconnect request
"Version"	"1.0"	Protocol version
"ApplicationName"	"<ApplicationName>"	Name of application, as sent in the connection request
"PID"	1234	Process identification number of application in Linux operating system, as sent in connection request

Disconnect handling

After receiving a disconnect request with application configuration data of a client application, the RIB_App processes the JSON string as follows:

- Checks if the JSON strings are a supported and valid message
  If the message is not supported or contains an error, the RIB_App does not process the string. In the response message, the RIB_App sends back an error message to the client application.
- Detects which application wants to disconnect from the RIB
- Removes all data related to this disconnecting application from its internal data store
- Identifies whether the disconnecting application provides symbols
  In this case, the RIB_App informs all client applications that consume these symbols about the removal of the symbol from the RIB. The RIB_App uses the following 'Provider Disconnect Info Message':

Key	Value	Description
"Type"	"ProviderDisconnectInfo"	Type of message; must be `ProviderDisconnectInfo` for 'Provider Disconnect Info Message'
"Version"	"1.0"	Protocol version
"RIBInformation""	"RibInformationData"	Contains all information regarding connection to RIB
"SymbolsToDisconnect"	"SymbolsToDisconnectData"	Contains all information regarding symbols that are not available anymore because provider applications are disconnected

"SymbolsToDisconnectData"

The "SymbolsToDisconnectData" contains at least one of the following objects:

Key	Value	Description
"<SharedMemoryId>"	["<SymbolName>", "<SymbolName>", ...]	List of the symbols that are disconnected.

The RIB_App expects that each notified client application acknowledges this message using a 'Provider Disconnect Response Message'.

When a consumer application receives a 'Provider Disconnect Info Message' from the RIB_App, it has to stop accessing the corresponding shared memory.

After a consumer has stopped accessing the shared memory of a provider, the consumer must acknowledge the provider disconnect notification to the RIB_App. This message may also contain the status for each shared memory that was closed. In order to do that, the optional attribute 'DisconnectStatus' containing a list of the closed shared memory names and a status is added.

This message has the following format:

Key	Value	Description
"Type"	"ProviderDisconnectRes-ponse"	Type of the message; must be `ProviderDisconnectResponse`
"Version"	"1.0"	Protocol version
"ApplicationName"	"<ApplicationName>"	Name of application that sends response message
"PID"	1234	PID of application in Linux operating system that sends response message
"Result"	"<OK or Error>"	Indicates whether reading from shared memories was stopped (`OK`) or not (`Error`)
"ErrorMessage"	"A detailed error message"	Optional: Detailed error message telling RIB why consumer cannot stop reading from shared memory
"DisconnectStatus"	"DisconnectStatusList"	Optional: A list of shared memory and its disconnect status

The elements of the 'DisconnectStatusList' are defined by the following attributes:

Key	Value	Description
"ShmID"	"<SharedMemoryName>"	Name of shared memory that should be closed by consumer
"Result"	"<OK or Error>"	Indicates whether reading from the shared memories was stopped (`OK`) or not (`Error`)

The 'DisconnectStatus' is defined as optional attribute and can be omitted by the user.

The RIB_App expects that this "ProviderDisconnectResponse" message is returned by the client application within a defined time. In case of a timeout or the shared memory could not be closed ('Result=="Error"') the RIB_App informs the provider about the not closed shared memories in the "DisconnectResponse" message.

Disconnect response

The RIB_App sends a disconnect response to a disconnecting application that is only consuming data, after it has been successfully removed from the RIB_App data store. In this case the consumer only client application has to wait for a 'Disconnect Response' of the 'RIB_App' before it can be closed.

The RIB_App sends a disconnect response to a disconnecting application that provides symbols in the RIB. In this case, the provider application sends a disconnect request to the RIB. The provider application waits for an answer from the RIB. The RIB sends this answer after all corresponding consuming applications have been informed and have acknowledged the disconnection of the provider.

The RIB_App disconnect response message is sent as a JSON formatted string and contains the following information:

Key	Value	Description
"Type"	"DisconnectFromRIB"	Type of message; must be `DisconnectFromRIB` for disconnect request
"Version"	"1.0"	Protocol version; should fit to current RIB version, for example `1.0`
"RIBInformation"	"RibInformationData"	Contains all information regarding the connection to RIB

"RibInformationData"

"RibInformationData" contains the following information:

Key	Value	Description
"RIBPid"	1111	Process identification number of RIB_App in Linux operation system
"RIBVersion"	"1.0"	Version of RIB_App, for example `1.0`
"Result"	"<Disconnected or Error>"	Indicates whether disconnection was successful (`Disconnected`) or not (`Error`)
"ErrorMessage"	"A detailed error message"	Optional: Detailed error message telling the user what went wrong in case of an error

Based on the result of the consumer application reply to the 'ProviderDisconnectInfo' message, the "RibInformationData" might contain different data and a different meaning for the client application.

The following scenarios are possible:

- All consumer applications stopped reading from the provider application's shared memory. Disconnection was successful and the application was removed from the RIB.
- One or more consumer applications did not respond to the RIB_App 'ProviderDisconnectInfo'. After a timeout, the disconnect response sent to the provider application contains the following error information:
  - The "Result" value of the "RibInformationData" is "Error".
  - The error message contains the string "Timeout occurred at:" followed by a listing of consumer applications that did not respond. This list contains the names of the applications and their known PID in the Linux operating system, for example "Timeout occurred at: ConsumerClientApp1 (1235); ConsumerClientApp2 (1236);"
    The providing client application is not allowed to delete the shared memory, because there might be consumers that still try to access the shared memory.
    Remove the shared memory manually.
    - Usage of POSIX shared memory: Delete the corresponding file in the Linux filesystem (/dev/shm) manually.
    - Usage of VMM shared memory: Execute the tool "s7_RIB_memory_cleaner".
    For more information on the "s7_RIB_memory_cleaner", refer to section Cleaning lifetime buffer (Page 77).

Note

If the Software Controller/CPU 1518(F)-4 PN/DP MFP is the provider of that shared memory, the shared memory is deleted and recreated automatically when the Software Controller/CPU 1518(F)-4 PN/DP MFP reconnects to RIB.

6.9 Programming example

The following programming example shows an example of a Consumer JSON file:

```
{
    "Type": "ConnectToRIBConfig",
    "Version": "1.0",
    "App_Sample_Consumer": {
        "Type": "ApplicationData",
        "Version": "1.0",
        "Description": "This app consumes a symbol named Symbol_1.",
        "Manufacturer": "Siemens AG",
        "PID": 12342,
        "Requests": {
            "Symbols": [
                "Symbol_1"
            ]
        }
    }
}
```

The following programming example shows an example of a Provider JSON file:

```
{
    "Type": "ConnectToRIBConfig",
    "Version": "1.0",
    "App_Sample_Provider": {
        "Type": "ApplicationData",
        "Version": "1.0",
        "Description": "This app provides a symbol named Symbol_1 of
type uint8_t.",
        "Manufacturer": "Siemens AG",
        "PID": 12341,
        "Provides": {
            "App_Sample_Provider_SHM": {
                "Type": "Provide",
                "Version": "1.0",
                "Description": "This shared memory only offers one
value",
                "Signal": -1,
                "CycleTimeInMicroseconds": 10000,
                "Symbols": {
                    "Symbol_1": {
                        "Offset": 0,
                        "Size": 1,
                        "Type": "uint8_t"
                    }
                }
            }
        }
    }
}
```

The following programming example shows an example of a Consumer and Provider JSON file:

```
{
    "Type": "ConnectToRIBConfig",
    "Version": "<MESSAGE_VERSION_TO_REPLACE>",
    "App_Sample_Provide_and_Consumer": {
        "Type": "ApplicationData",
        "Version": "1.0",
        "Description": "This app provides a symbol named Symbol_1 of
type uint8_t and consumes a symbol named Symbol_2.",
        "Manufacturer": "Siemens AG",
        "PID": 12343,
        "Provides": {
            "App_Sample_Provide_and_Consumer": {
```

```
                "Type": "Provide",
                "Version": "1.0",
                "Description": "This shared memory only offers one
value",
                "Signal": -1,
                "CycleTimeInMicroseconds": 10000,
                "Symbols": {
                    "Symbol_1": {
                        "Offset": 0,
                        "Size": 1,
                        "Type": "uint8_t"
                    }
                }
            }
        },
        "Requests": {
            "Symbols": [
                "Symbol_2"
            ]
        }
    }
}
```

RIB Support Library

7

7.1 Introduction

The RIB Support Library is a high-level C++-API (Application Programming Interface) for writing RIB client applications.

The RIB Support Library makes it easy to connect to RIB, manages symbols and reads/writes them.

The library consists of one library shared object file and C++ header files. The main entry point for the developer is the 'RibClient' class and its related objects. All these objects can be included by the single header file 'ribClient.h'. The RIB Support Library also uses the SIMATIC VMM Library to access a hypervisor-based shared memory.

The RIB Support Library has the following design features:

- The API must be written in C++
- The C++ standard used is C++17 and the compiler (GNU Compiler Collection) used is GCC10.2.1(libstdc++ 6.0.28, glibc 2.31-13)
- You can configure a RIB client object using a configuration string in a defined JSON format

---

Note

Thread safety of objects created by RIB Support Library

Instances of the 'RibClient', 'Reader' and 'Writer' and other related objects are not thread-safe by default. In a multi-threaded application, it is the responsibility of the developer to ensure thread safety.

Receiving symbol information after activation

The first release of the RIB Support Library did not support changes of the availability of consumed symbols after activation was done. Due to its blocking behavior it is waiting for all requested symbols to be available on activation. This has changed in the new version of the RIB Support Library (RIB V2.0).

With this library version, even after activation, the consumer receives the information from the RIB_App about symbols that become available or unavailable every time a provider connects or disconnects. The information about available symbols can be obtained any time by the user simply by requesting a new `SymbolToPointerMap` that includes the new symbols and their pointers using the method `SymbolToPointerMap symbolMap = reader->getSymbolNameToPointerMap();` . Also a call to `getPointerToSymbol()` returns the newly available pointer.

7.2 Connecting a client to RIB

RibClient

To connect a client to RIB, an instance of 'RibClient' must be created. The 'RibClient' object is configured by default, but you can set your own configuration data, for example IP-address, application name or configuration string:

```
RibClient::RibClient ribClient;

auto result = ribClient.setIPAddress("127.0.0.1");
if (result != RibReturnCode::OK)
{
    // error handling here
}

string configurationData = ...;
result = ribClient.setConfigurationData(configurationData);
if (result != RibReturnCode::OK)
{
    // error handling here
}

result = ribClient.setApplicationName("MyApplication");
if (result != RibReturnCode::OK)
{
    // error handling here
}
```

A detailed example of `string configurationData` can be found in section Example configuration string (Page 92).

Logging

The RIB Support Library supports a configurable logging mechanism that logs data to the standard output. The 'RibClient' class offers the following static method:

```
static void setLogLevelToDebug(const bool enable);
```

This method changes the detail level of the log messages. The method has the following characteristics:

- By default, the logged data only contains necessary information, for example important information and errors
- When "enable" is set to "true", additional log data is printed to "stdout", for example debugging information
- When "enable" is set to "false", the additional output is suppressed

Retrieve library version

You can retrieve the currently used version of the RIB Support Library. To retrieve the currently used version, call the following method:

```
std::string getLibraryVersion();
```

The version should match the version of the currently used 'RIB_App'.

RibReturnCodes

For a list of all RibReturnCodes, refer to section RibReturnCode (Page 94).

7.3 Defining symbols

When RIB is configured, the 'RibClient' object needs to know all consumed and provided symbols. Use the methods 'InitRequestedData' and 'InitProvidedData'.

Adding provided symbols to configuration

A provider can use the method below to provide the configuration data object with the following information:

- name of shared memory
- provided symbols
- cycle time in microseconds

```
RIBClient::initProvidedData(lifetimeBufferName, symbolList,
cycleTimeInMicroSeconds)
```

A RibReturnCode is returned and must be handled by the user:

```
const std::string& lifetimeBufferName = ...;
const std::list<RIB::SymbolDescription> symbolList = {
  { "myValue_x", RIB::SymbolDescription::DataType::UINT64_T, 1},
  { "myValue_y", RIB::SymbolDescription::DataType::UINT64_T, 1} };
const std::uint64_t cycleTimeInMicroSeconds = 500;
result = ribClient.initProvidedData(lifetimeBufferName, symbolList,
cycleTimeInMicroSeconds);
if (result != RibReturnCode::OK)
{
    // error handling here
}
```

Forcing deletion of lifetime buffer before creating new lifetime buffer

When initializing provided data, a name for the shared memory of the lifetime buffer is specified. If a shared memory with this specific name already exists, the method `RibClient::activate()` returns the following error message:

`RibReturnCode::GenerateLifetimeBufferFailed`

To avoid this error message, make sure to clean up the system. To clean up the system, call the method `RibClient::deactivate()` and the destructor of the 'RIBClient' objects.

The 'RibClient' offers the following function to force the deletion of an existing lifetime buffer:

```
setForceDeletingExistingLifetimeBuffer(std::string
lifetimeBufferName)
```

The name of a lifetime buffer must be unique. If a lifetime buffer with such a name already exists, this existing lifetime buffer will be deleted before creating a new one.

Note

Deleting existing shared memory containing a lifetime buffer

Deleting existing shared memory that contains a lifetime buffer might lead to undefined behavior, if the deleted shared memory is still used by consuming or providing applications.

Adding requested symbols to configuration

Use the following API call to register symbol names:

```
RibClient::initConsumedData(const std::string& symbolName)
```

The following RibReturnCode must be handled by the user:

```
const auto result = ribClient.initConsumedData("ASymbolToRead");
if (result != RibReturnCode::OK)
{
    // error handling here
}
```

The RibClient can now communicate and read/write data from/to the RIB.

initConsumedData for a list of symbol names

In addition to the already existing function `RibReturnCode RibClient::initConsumedData(const std::string&)`, an overloaded method `RibReturnCode RibClient::initConsumedData(const std::list<std::string>& symbolNames)` is added that takes a list of symbol names to consume.

```
// 1. Initialize the RibClient object

RIB::RibClient ribClient;


// 2. Configure RIB consumer part

const std::list<std::string> symbolToBeConsumed = {"switch1",
"switch2", "switch3"};
```

```
ribClient.initConsumedData(symbolToBeConsumed);
```

If one of the symbol names to be added has already been added before, it will not be added a second time.

RibReturnCodes

For a list of all RibReturnCodes, refer to section RibReturnCode (Page 94).

7.4 Activating client

After configuring the provided and/or consumed symbols, use the following method to activate the client:

```
RibClient::activate()
```

This method carries out the following steps:

1. Creates the shared memory for providing symbols
2. Connects the application to RIB
3. Registers the configured provided and/or consumed symbols

The following RibReturnCode must be handled by the user:

```
const auto result = ribClient.activate();
if (result != RibReturnCode::OK)
{
    // error handling here
}
```

The RibClient can now communicate with the RIB and data exchange is available for consuming and/or providing operations.

RibReturnCodes

For a list of all RibReturnCodes, refer to section RibReturnCode (Page 94).

7.5 Reading data from lifetime buffer

Accessing the reader object

To read data from the lifetime buffer of a provider, a consumer needs to obtain a reader object via the following method:

```
std::tuple<RibReturnCode, std::shared_ptr<Reader>>
RibClient::getReader()
```

The first element of the returning tuple is a RibReturnCode that must be handled by the user. The second element of the returning tuple is a smart pointer. This pointer owns a 'Reader' object through the pointer that allows reading data from a lifetime buffer.

```
auto [getReaderResult, reader] = ribClient.getReader();
if (getReaderResult != RibReturnCode::OK)
{
       // error handling here
}
```

Accessing the symbols for reading their values

With the reader object generated by calling `getReader`, a consumer application is able to place a pointer to a symbol via the following method:

```
std::pair<RibReturnCode, T*>getPointerToSymbol<T>(const
std::string& symbolName)
```

The template parameter 'T' represents one of the allowed data types. The first element of the returned pair is a RibReturnCode that must be handled by the user. The second element of the returned tuple is a raw pointer for accessing the symbol and reading its value.

```
auto [getPointerResult, ptrToSymbol] = reader-
>getPointerToSymbol<std::uint64_t>("ASymbolToRead");
if (getPointerResult != RibReturnCode::OK)
{
       // error handling here
}
```

Note

The parameter type must match the provided symbol type. The system does not check whether these two types match.

Make sure to use the correct type.

You can additionally obtain a list of all requested symbols and their corresponding pointer. So only a single method call is needed.

This method is added to the class `RIB::Reader`. It returns a map containing the symbol name as key and the pointer to the value. These pointers are `nullptr` in case the symbol is not available yet. Therefore, it can be used the following way:

```
// get the reader
auto [getReaderResult, reader] = ribClient.getReader();
...
// get the maps containing the symbol name as key and the pointer
as value
SymbolNameToPointerMap symbolMap = reader-
>getSymbolNameToPointerMap();
// access the value for a given symbol named "Symbol_1"
void const* ptrToSymbol_1 = symbolMap["Symbol_1"];
if (ptrToSymbol_1 != nullptr)
{
  auto value = *static_cast<int*>(ptrToSymbol_1);
  std::cout << "Value of Symbol_1=" << value << std::endl;
}

// access all symbols of the map and interpret them as int
for (const auto& [symbolName, ptrToSymbol] : symbolMap)
{
  if (ptrToSymbol != nullptr)
  {
    auto value = *static_cast<int*>(ptrToSymbol);
    std::cout << "Value of " << symbolName << "=" << value <<
std::endl;
  }
}
```

Continue reading data even if provider is disconnected

In the first version of the RIB Support Library the consumer application stops reading from all connected providers, if one of the providers was disconnected from RIB. Reading from the remaining shared memories of other providers was not possible and the consumer application stopped working.

In the new version of the RIB Support Library (RIB V2.0) only the shared memory of the disconnected provider is closed. The shared memories and symbols of the remaining connected providers are still available and their data values can be read by the consumer.

The method `read()` returns a tuple consisting of a RibReturnCode and an updateAvailable flag. The updateAvailable flag is set to `true`, when a provider connects or disconnects until the method `reader->getSymbolNameToPointerMap()` is called. This resets updateAvailable to `false`.

Note

Also a call to `reader->getPointerToSymbol(...)` resets the internal updateAvailable flag to `false` as this method always provides the latest pointer address.

Even if a provider disconnects, `RibReturnCode::OK` indicates that data from other connected providers can still be read successfully by the consumer application.

As long as you do not call `SymbolImage::getSymbolNameToPointerMap()`, the pointers belonging to the disconnected shared memory and symbols remain valid and contain the last read value. Pointers that were accessed using the method `reader->getPointerToSymbol(...)` also remain valid and contain the last read value until `SymbolImage::getSymbolNameToPointerMap()` is called or the reader object is released.

You can update the pointers used in your application by calling the method `SymbolImage::getSymbolNameToPointerMap()` . With this call, all previously disconnected shared memories and symbols are not available anymore in the client application. The related pointers must not be accessed anymore in the user program. Make sure to update your locally used pointer references in this case to prevent segmentation faults.

Note

Recommendation

Do not mix `SymbolImage::getSymbolNameToPointerMap()` and `reader->getPointerToSymbol(...)` in your code. Pointers aquired with `reader->getPointerToSymbol(...)` become invalid when `SymbolImage::getSymbolNameToPointerMap()` cleans up old reader objects that are not in use anymore.

Receive symbol update information after activation

Due to the "non-blocking activate" behavior and the need for a consumer to continue working even after another provider disconnects, there is a need to frequently update the list of symbol pointers by calling `reader->getSymbolNameToPointerMap()` in each loop of the user program. This however might cause performance issues.

To reduce the workload by not requesting the `SymbolNameToPointerMap` in each loop the `Reader::read()` method was changed. It now returns in addition to the "RibReturnCode" an "updateAvailable" flag. This flag is `true` in case the content of the `SymbolNameToPointerMap` has changed since the last call to `reader-`

`>getSymbolNameToPointerMap()`. So you can update his `SymbolNameToPointerMap` on changes only.

```
// get the reader
auto [getReaderResult, reader] = ribClient.getReader();
...
// get the maps containing the symbol name as key and the pointer
as value
SymbolNameToPointerMap symbolMap = reader-
>getSymbolNameToPointerMap();

while (loop)
{
  // check for changes in SymbolNameToPointerMap
  auto [errorCode, updateAvailable] = reader->read();

  if (updateAvailable)
  {
    // only get new copy of SymbolToPoinerMap
    symbolMap = reader->getSymbolNameToPointerMap();
  }

  for (const auto& [symbolName, ptrToSymbol] : symbolMap)
  {
    if (ptrToSymbol != nullptr)
    {
      auto value = *static_cast<int*>(ptrToSymbol);
      std::cout << "Value of " << symbolName << "=" << value <<
std::endl;
    }
  }
}
// access all symbols of the map and interpret them as int
```

Note

Updating a large amount of symbol pointers

Updating a large amount of symbol pointers in the application loop, in case `updateAvailable` has changed to `true` when
reading data, might take a longer time and needs to be handled with care to avoid violating cycle times or timing behavior.

Read operation

The method `read()` of the reader object executes a read operation for all acquired symbol values from the RIB applications' shared memory. Afterwards the symbol values can be used in the consumer program.

The following RibReturnCode is returned that must be handled by the user.

```
auto retVal = reader-> read();
if (retVal == RibReturnCode::OK)
{
    std::cout << "ASymbolToRead=" << std:to_string(*ptrToSymbol) <<
std::endl;
}
else if (retVal == RibReturnCode::BufferNotWrittenByProducer)
{
    // special case provider has not written data yet
}
else
{
    // error handling here
}
```

Possible error return codes:

- If `RibReturnCode::DataNotAvailable` is returned, any of the connected providing applications has been terminated. In this case, we recommend deactivating the connection. If you want to restart the provider, reactivate the existing connection. To reactivate the existing connection, the reader and writer objects must be generated again.
- If `RibReturnCode::ReadTimeOut` is returned, reading the data from one lifetime buffer takes longer than the specified segment lifetime. If this occurs once, we recommend that you repeat the read operation. If this occurs frequently, reduce the system utilization or increase the process priority of the reading application.
- If `RibReturnCode::BufferNotWrittenByProducer` is returned, the provider has not written any data yet. In this case, we recommend that you repeat the read operation.

- The error code `RibReturnCode::InvalidBufferElement` is caused by the providing application. This error cannot be corrected by the consumer application.
- The error code `RibReturnCode::ReadError` indicates that accessing the reader failed and that the lifetime buffer is not available for reading. Possible reasons for the error might be that the provider has been stopped or the shared memory has been deleted.

RibReturnCodes

For a list of all RibReturnCodes, refer to section RibReturnCode (Page 94).

7.6 Writing data to lifetime buffer

The RIB Support Library provides a 'Writer' class. The 'Writer' class grants write data access to RIB lifetime buffers based on the realtime data exchange concept.

Accessing the writer object

To write data into the lifetime buffer, a provider needs to obtain a writer object via the following method:

```
std::tuple<RibReturnCode, std::shared_ptr<Writer>>
RibClient::getWriter(const std::string& sharedMemory)
```

The parameter `lifetimeBufferName` represents the name of the lifetime buffer already used when defining this buffer with `InitProvidedData()`.

The first element of the returned tuple is a RibReturnCode that must be handled by the user. The second element of the returned tuple is a smart pointer. This pointer owns a 'Writer' object through the pointer that allows writing data to a lifetime buffer.

```
auto [getWriterResult, writer] =
ribClient.getWriter(lifetimeBufferName);
if (getWriterResult != RibReturnCode::OK)
{
    // error handling here
}
```

Accessing symbols for writing their values

After obtaining the writer object by calling `getWriter`, a provider application is able to direct a pointer to a symbol using the following method:

```
std::pair<RibReturnCode, T*>getPointerToSymbol<T>(const
std::string& symbolName)
```

The template parameter 'T' represents one of the allowed data types. The first element of the returned pair is a RibReturnCode that must be handled by the user. The second element of the returned tuple is a pointer to access the symbol and set its value.

```
auto [getPointerResult, ptrToSymbol] = writer-
>getPointerToSymbol<std::uint64_t>("ASymbolToWrite");
if (getPointerResult != RibReturnCode::OK)
{
    // error handling here
}
```

Note

The system does check whether the type provided here matches the type given in the initialization of the symbol description when initializing the provider.

Make sure to use the correct type.

You can additionally obtain a list of all requested symbols and their corresponding pointer, so only a single method call is needed.

This method is added to the class `RIB::Writer`. It returns a map containing the symbol name as key and the pointer to the value. These pointers are available immediately after successful creation of the `RIB::Writer` object and will not be `nullptr`. It can be used the following way:

```
// get the writer

auto [getWriterResult, writer] = ribClient.getWriter();

...

// get the maps containing the symbol name as key and the pointer
as value

SymbolNameToPointerMap symbolMap = writer-
>getSymbolNameToPointerMap();


while (!stopProcess)

{

  // write data to the shared memory

  retVal = writer->write();

  if (retVal != RibReturnCode::OK)

  {

    return EXIT_FAILURE;

  }
```

```
  // access all symbols of the map and interpret them as int and
increment them
  for (auto& [symbolName, ptrToSymbol] : symbolMap)
  {
    const auto value = ++(*static_cast<int*>(ptrToSymbol));
    std::cout << "New value of " << symbolName << "=" << value <<
std::endl;
  }
}
```

Write operation

With the writer object, a provider is able to write its data into the shared memory using the `write()` method.

The following RibReturnCode must be handled by the user.

```
auto retVal = writer->write();
if (retVal != RibReturnCode::OK)
{
    // error handling here
}
```

Possible error return codes:

- The error code `RibReturnCode::WriteSymbolsInvalidParameter` can not be influenced by the user.

RibReturnCodes

For a list of all RibReturnCodes, refer to section RibReturnCode (Page 94).

7.7 Deactivating client

A RibClient object allocates several resources, e.g. for:

- Maintaining an open socket connection to the RIB_App
- Accessing lifetime buffers in shared memories

Note

We strongly recommend that you free these resources by using the following method:

`RibClient::deactivate()`

This method closes the connection to the RIB, cleans up the lifetime buffers' and shared memory accesses used for reading data. The lifetime buffers' and shared memory accesses used for writing data are cleaned up by destroying the RibClient object.

The following RibReturnCode can be handled by the user.

```
retVal = RibClient::deactivate()
if (retVal != RibReturnCode::OK)
{
    // error handling here
}
```

If the deactivation of a provider is not acknowledged by the RIB_App (e.g. if the connection to the RIB_App was terminated or a consumer does not respond), the 'Lifetime Buffers' will not be closed.

Possible error return codes:

- If `RibReturnCode::NotConnected` is returned, the RibClient has not been activated before or activation failed.
- The error code `RibReturnCode::SocketCommunicationError` is an internal error that cannot be influenced by the user.
- If `RibReturnCode::SignOutTimeOut` is returned, at least one of the consuming applications has not acknowledged the deactivation within the waiting time specified by the RIB. Either check the consumers or increase the RIB_App waiting time using parameter `-w`. If the application is a provider, it will not delete its lifetime buffer. For more information, refer to chapter Cleaning lifetime buffer (Page 77).
- The error code `RibReturnCode::SignOutUnknownError` is an internal error

RibReturnCodes

For a list of all RibReturnCodes, refer to section RibReturnCode (Page 94).

7.8 Obtaining a string for RibReturnCode

The following RibClient member function returns a string containing the predefined description of a given RibReturnCode:

```
getRibReturnCodeAsString(const RibReturnCode code)
```

You can use this function to obtain a description of occurring errors, for example, logging errors during the execution of an application.

RibReturnCodes

For a list of all RibReturnCodes, refer to section RibReturnCode (Page 94).

7.9 Registering and unregistering signal callback functions

To register a callback function to a signal, use the following method:

```
registerCallback(int posixSignal, const std::function<void(int)>
callback)
```

If an application needs to be terminated when a signal is received, the corresponding signal has to be registered here. This is necessary in order to properly terminate the connection to the RIB in the background.

To terminate the client when receiving the specified signals, use this method instead of registering signals directly. If you do not want to use a signal for process termination, do not use this method.

You can use this method for all POSIX (Portable Operating System Interface) signals except SIGKILL and SIGSTOP. For signal registration, this method grants multiple client registrations to one POSIX signal within one process. If the callback parameter is invalid, this function might throw a RIBException error.

The register method returns a cookie identifier. To later unregister from the signal, use this identifier with the following method:

```
unregisterCallback(int cookie)
```

If the cookie parameter is an integer below zero, the method might throw a RIBException error.

Note

Make sure to use the correct cookie identifier to avoid unregistering from a wrong signal.

---

8 Programming using the SFC ODKP_ISC (SFC65490)

8.1 Configuration

For sharing data between the CPU and Linux applications, the SFC65490 establishes a connection to RIB.

The SFC offers the following five modes for carrying out all RIB-related operations:

- "OP_INIT"
- "OP_CONNECT"
- "OP_READ"
- "OP_WRITE"
- "OP_DISCONNECT"

The SFC ODKP_ISC is not part of the TIA Portal hardware catalog. You can add the SFC by searching ODKP_ISC by its name in TIA Portal.

Requirements

For using RIB and SFC, the following requirements apply:

- You can only use RIB on IndOS operating systems, a Siemens Hypervisor and shared memory.
- RIB does not support Windows operating systems.
- You cannot run multiple SFC calls at the same time. The system must first finish an ongoing operation, before handling a new operation.

If one or multiple of these mentioned conditions are not met, the following SFC return codes appear:

Status	Error reason
0x8090	The given operation ID is not supported. Only the listed five modes are supported.
0x809B	Hypervisor shared memory does not exist or the SFC is used in a Windows variant.
0x80C3	The SFC is already in progress. The SFC cannot be called concurrently.

SFC return codes

Preconditions

Before initializing operation, carry out the following steps:

For requested symbols:

1. Create a user data type (UDT) for the requested symbol.
2. Create a type DB with the user data type created in the previous step.
3. Create a DB_ANY variable in another DB and assign this consumed type DB to the variable.

For provided symbols:

1. Create a user data type (UDT) for the provided symbol.
2. Create a type DB with the user data type created in the previous step.
3. Create a DB_ANY variable in another DB and assign this provided type DB to the variable.
4. Create an unsigned long integer variable for the cycle time in another DB.

Note

Optimized DBs

For the created DBs, the attribute "Optimized block access" must be enabled.

Note

Multiple SFC operations

You cannot call multiple SFC operations at the same time. The system must first finish an ongoing operation, before handling a new operation.

The steps to be followed as preconditions are described in chapter "OP_INIT" mode (Page 64).

8.2 "OP_INIT" mode

8.2.1 "Init" operation

The initialize ("Init") operation configures the provided and consumed symbols. You can only use the "Init" operation with a rising edge. The "Init" operation is called only once. You cannot change the symbol configuration after its execution. If you need to reconfigure symbols, call the "disconnect" operation first. Then modify the configuration and call the "Init" operation again.

Restrictions

Calling the "Init" operation is subject to the following restrictions:

- **Consumed DBs and provided DBs**
  The maximum number of supported consumed DBs and provided DBs is 16 each.
  If you use more than one consumed DB and/or provided DB, you must create a DB_ANY Array variable. Then assign all DBs to this variable.
- **Cycle time**
  You can use a common cycle time for all provided DBs.
  If you only set one cycle time value, then this value applies to all DBs. If you set different cycle times for each DB, the array index of the provided DB must match the array index of the cycle time.

- **Names of DBs**
  The names of provided and consumed DBs are used as lifetime buffer names. Make sure that each of these DB names is unique. Also make sure that these DB names are not the same as lifetime buffer names created by Linux applications.
- **Symbols for consumed DBs/provided DBs**
  The maximum number of symbols for consumed DBs and provided DBs is 1024 each.
- **Symbol names**
  Symbol names must be unique in the whole RIB ecosystem, otherwise RIB returns an error when calling the "connect" operation.

Supported data types

For the RIB environment, the CPU supports reading and writing the following data types.

TIA Portal	C++	Size in Bytes
ULINT	uint64_t	8
UDINT	uint32_t	4
UINT	uint16_t	2
USINT	uint8_t	1
LINT	int64_t	8
DINT	int32_t	4
INT	int16_t	2
SINT	int8_t	1
LREAL	double	8
REAL	float	4
BYTE	uint8_t	1

Data type conversion table

You can also use an array of these supported data types as symbol. An array of the supported data types as symbol counts as only one symbol.

Return codes of STATUS variable

Calling the "Init" operation may return the following codes in the STATUS variable.

STATUS	Reason
0x0000	"Init" operation successful
0x7000	No active job
0x8154	Invalid data type for ParPtr1
0x8254	Invalid data type for ParPtr2
0x8354	Invalid data type for ParPtr3
0x8454	Invalid data type for ParPtr4
0x8281	Invalid value for ParPtr2
0x8381	Invalid value for ParPtr3
0x8481	Invalid cycle time value given. Cycle time must be greater than 0. Cycle time array and provided DB array mismatch.

STATUS	Reason
0x80A2	Internal error Contact customer support.
0x80A3	Unsupported data type
0x80AA	Maximum number of supported symbols exceeded
0x80AB	No provided or consumed DBs given. ParPtr2 and ParPtr3 are both NULL.
0x80AC	Maximum number of supported DBs is exceeded
0x80AD	Failsafe DBs are not supported
0x80B1	Not enough resources for JSON allocation
0x80D3	One of the provided DBs is registered more than once

Return values of OP_INIT mode

8.2.2 ODKP_ISC "Initialize" mode parameters

ODKP_ISC "Initialize" mode parameters

The following example shows the parameters of the OP_INIT mode for the consumer and provider configuration of the Software Controller.

Code	Comments
`#Result_Init := ODKP_ISC(OperationID := "OP_INIT",`	`// Initialize RIB symbols for requested and provided DBs`
`    ParPtr1 := "RIB".InitRequest,`	`// Rising edge control for executing the operation`
`    ParPtr2 := "ConsumedDBArray",`	`// Consumed DB(s) (optional)`
`    ParPtr3 := "ProvidedDBArray",`	`// Provided DB(s) (optional)`
`    ParPtr4 := "RIB.CycleTime,`	`// Cycle time for provided symbols update (mandatory, if ParPtr3 is set for a provided DB)`
`    ParPtr5 := NULL);`	

The following table gives an overview of the available OP_INIT mode parameters and their characteristics.

Section	Name	Data type	Description
Automatically generated parameters by ODKP_ISC			
Output	STATUS	INT	Function result error message
User defined parameters			
Input	OperationID	INT	OP_INIT (id: 0)
InOut	ParPtr1	BOOL	Rising edge control variable to execute "Init" operation; when input signal goes from low (0) to high (1)

Section	Name	Data type	Description
InOut	ParPtr2	DB_ANY or DB_ANY Array	Requested DB(s)
InOut	ParPtr3	DB_ANY or DB_ANY Array	Provided DB(s)
InOut	ParPtr4	ULINT or ULINT Array	Scheduled update cycle interval for provided DBs (in microseconds)
InOut	ParPtr5	NULL	reserved

OP_INIT mode parameters

ODKP_ISC "Initialize" mode parameters for consumer only configuration

The following example shows the parameters of the OP_INIT mode for a consumer-only configuration of the Software Controller.

Code	Comments
`#Result_Init := ODKP_ISC(OperationID := "OP_INIT",`	`// Initialize RIB symbols for requested DBs`
`    ParPtr1 := "RIB".InitRequest,`	`// Rising edge control for executing the operation`
`    ParPtr2 := "ConsumedDBArray",`	`// Consumed DB(s) (mandatory since SWCPU is consumer only app)`
`    ParPtr3 := NULL,`	
`    ParPtr4 := NULL,`	
`    ParPtr5 := NULL);`	

The following table gives an overview of the available OP_INIT mode parameters and their characteristics.

Section	Name	Data type	Description
Input	OperationID	INT	Parameter must be set to 0, which is OperationID for "Initialize"
InOut	ParPtr1	BOOL	Rising edge control variable to execute "Init" operation
InOut	ParPtr2	DB_ANY or DB_ANY Array (mandatory)	for consumed DB(s)
InOut	ParPtr3	NULL	reserved
InOut	ParPtr4	NULL	reserved
InOut	ParPtr5	NULL	reserved

ODKP_ISC "Initialize" mode parameters for provider-only configuration

The following example shows the parameters of the OP_INIT mode for a provider-only configuration of the Software Controller.

Code	Comments
`#Result_Init := ODKP_ISC(OperationID := "OP_INIT",`	`// Initialize RIB symbols for provided DBs`
`    ParPtr1 := "RIB".InitRequest,`	`// Rising edge control for executing the operation`
`    ParPtr2 := NULL,`	
`    ParPtr3 := "ProvidedDBArray",`	`//Provided DB(s) (mandatory since SWCPU is a provider only app)`
`    ParPtr4 := "RIB.CycleTime,`	`//Cycle time for provided symbols (in microseconds)`
`    ParPtr5 := NULL);`	

The following table gives an overview of the available OP_INIT mode parameters and their characteristics.

Section	Name	Data type	Description
Input	OperationID	INT	Parameter must be set to 0, which is OperationID for "Initialize"
InOut	ParPtr1	BOOL	Rising edge control variable to execute "Init" operation
InOut	ParPtr2	NULL	reserved
InOut	ParPtr3	DB_ANY or DB_ANY Array (mandatory)	for provided DB(s)
InOut	ParPtr4	Long Integer or Long Integer Array (mandatory)	Scheduled update cycle interval for provided DBs (in microseconds) The array index of the provided DB must match the array index of the cycle time.
InOut	ParPtr5	NULL	reserved

Impact of Initialize operation on performance

The Initialize operation internally resolves, extracts and decomposes all DB symbols and retrieves their offset, size and type information. For these reasons, the Initialize operation impacts the performance based on the data size of DBs and the variable count.

Note

Cycle time

Note that a high number of symbols to be initialized has an impact on the OB cycle time.

If your project handles large amounts of data or symbol counts, increase the maximum cycle time in the CPU configuration.

Also note that a high number of symbols can result in cycle times of some milliseconds and should therefore not be used in time critical OBs.

The Initialize operation is only executed once before connecting to RIB. You can use the operation in the startup OB. When reconfiguring the "Init" operation, a STOP-RUN transition of the CPU is necessary.

8.3 "OP_CONNECT" mode

8.3.1 Connecting to RIB

Before reading and writing any symbols, you must establish a connection to RIB. The connect operation handles the communication part with RIB. After establishing a RIB connection, you can access the CPU for shared memory operations.

Establish RIB connection

You establish a connection to RIB by using the mode "OP_CONNECT" (ID: 1) of ODKP_ISC (SFC65490). This mode starts an asynchronous connection request to the RIB_App.

Preconditions

- Before you connect to RIB, call an "OP_INIT" operation.
  For more information on how to call an "OP_INIT" operation, refer to section "OP_INIT" mode (Page 64).
- The port number used as a parameter in the OP_INIT operation must be the same as the port number configured for the RIB_App. The default value is 27567.
- The IP address used as a parameter for the OP_INIT operation must be the VNIC (virtual network interface card) IP address on IndOS side.

Calling the connect operation

The following example shows the parameters of the ODKP_ISC "Connect" mode.

Code	Comments
`#Result_Connect := ODKP_ISC(OperationID := "OP_INIT",`	
`ParPtr1 := "RIB".ConnectRequest,`	`// Rising edge control for executing the operation`
`ParPtr2 := "RIB.Address",`	`// TADDR_PARAM which has IP and port information of RIB`
`ParPtr3 := NULL,`	
`ParPtr4 := NULL,`	
`ParPtr5 := NULL);`	

The following table gives an overview of the available OP_CONNECT mode parameters and their characteristics.

Section	Name	Data type	Description
Automatically generated parameters by ODKP_ISC			
Output	STATUS	INT	Function result error message
User defined parameters			
Input	OperationID	INT	OP_CONNECT (id: 1)
InOut	ParPtr1	BOOL	Rising edge control variable to execute CONNECT operation
InOut	ParPtr2	TADDR_PARAM	IP address of Linux VNIC IP address and port information of RIB_App
InOut	ParPtr3	NULL	reserved
InOut	ParPtr4	NULL	reserved
InOut	ParPtr5	NULL	reserved

Parameters of OP_Connect mode

If you enter the provided DBs, the operation "ODKP_ISC CONNECT" writes the initial values of the provided symbols. If there is no initial value, the provided symbol values are filled up by zeros.

The CPU creates a lifetime buffer for the provided symbols.

The connect operation works on the rising edge. This means the operation is executed when the RIB connect request changes from FALSE to TRUE.

Return codes of STATUS variable

You can check the connection status with its return code. When the RIB_App crashes or closes, it can be seen from the connect operation’s return code.

Note

RIB_App crashes or closes

If the RIB_App is gone while the CPU continues to exchange data, consistent data transfer is no longer guaranteed due to the absence of RIB_App. When you restart the RIB_App, the CPU is not automatically reconnected. You must call the "disconnect" operation and then "init" operation for connecting to the RIB_App again.

Calling the "Connect" operation may return the following codes in the STATUS variable.

STATUS	Reason
0x0000	Connection request successful
0x7000	No active job
0x7001	Connection in progress
0x7002	Connection in progress already
0x8154	Invalid data type for ParPtr1
0x8254	Invalid data type for ParPtr2
0x80A0	"Init" operation must be executed first

STATUS	Reason
0x80A2	Internal error Contact customer support.
0x80A7	Requested DB not registered before with "Init" operation
0x80A8	Type of requested symbol(s) different from provided type
0x80B0	RIB returns "application name already exists"
0x80B3	Software Controller version is not compatible with RIB version
0x80B4	Response string does not match valid format of RIB configuration values
0x80B6	Lifetime buffer name exceeds 128 characters
0x80B7	RIB returns "invalid argument response"
0x80B8	RIB returns "attribute is missing" error
0x80B9	RIB returns "symbol type is different from the provided type"
0x80BA	RIB returns "provided symbol has been provided by a different provider"
0x80D1	Lifetime buffer provided by a Linux application could not be found
0x80D2	Lifetime buffer cannot be created on VMM (Virtual Machine Manager) shared memory
0x80F1	Connection request failed or RIB_App crashed

Return values of OP_CONNECT mode

8.4 "OP_READ" mode

8.4.1 Reading data from Linux applications

After successful connection and configuration, the CPU can start exchanging data.

Reading data

For reading data provided by Linux applications, use ODKP_ISC with "OP_READ" (ID: 3).

Preconditions

Before you call read operations, make sure that you have carried out the following steps:

1. You have called "OP_INIT" successfully.
   For more information on "OP_INIT", refer to section "OP_INIT" mode (Page 64).
2. You have called "OP_CONNECT" successfully.
   For more information on "OP_CONNECT", refer to section "OP_CONNECT" mode (Page 69).

Calling the read operation

The following example shows the parameters of the ODKP_ISC "Read" mode.

Code **Comments**

```
#Result_Read := ODKP_ISC(OperationID := "OP_READ",
    ParPtr1 := "RIB".EnableRead,                    // Enable read operation
    ParPtr2 := "SWCPU_DataExchange_DB",             // Requested DB
    ParPtr3 := NULL,
    ParPtr4 := NULL,
    ParPtr5 := NULL);
```

The following table gives an overview of the available OP_READ mode parameters and their characteristics.

Section	Name	Data type	Description
Automatically generated parameters by ODKP_ISC			
Output	STATUS	INT	Function result error message
User defined parameters			
Input	OperationID	INT	OP_READ (id: 3)
InOut	ParPtr1	BOOL	Enabler for executing READ operation
InOut	ParPtr2	Type DB	Requested DB
InOut	ParPtr3	NULL	reserved
InOut	ParPtr4	NULL	reserved
InOut	ParPtr5	NULL	reserved

Parameters of OP_READ mode

The OP_READ mode of ODKP_ISC supports a maximum of 16 DBs. The OP_READ mode can be called multiple times with different consumed DBs.

The CPU can read symbols provided by different Linux applications even though they are all stored in the same DB.

If one of the provider applications has disconnected from the RIB environment, the READ operation result value changes from SUCCESS(0) to REQUESTED_SYMBOLS_MISSING(0x7300). When the provider application reconnects, the CPU will continue to read and the result value will return to SUCCESS(0).

The read operation only works with the enabler state being TRUE.

The same DB must be used for both "init" and "read" operations. Changes in the DB (adding or deleting symbols) is not allowed.

RIB V1 does not support download in RUN.

If you use a cyclic OB for a read operation, set the OB priority to 16 or higher.

If every symbol in that requested DB is available, the READ operation starts immediately. If there are more than one DB, missing symbols in other DBs do not block that specific DB's read operation.

Return codes of STATUS variable

Calling the "read" operation might return the following codes in the STATUS variable.

STATUS	Reason
0x0000	Read operation successful
0x7000	No active job
0x7001	Provider application starts disconnecting while operation is running
0x7300	Requested symbols missing
0x7400	Data not published yet by provider application
0x8154	Invalid data type for ParPtr1
0x8254	Invalid data type for ParPtr2
0x80A2	Internal error Contact customer support.
0x80A4	Connect operation not finished yet
0x80A5	Read data outdated. Symbol could not be read in buffer element lifetime duration.
0x80A7	Requested DB not registered before with "Init" operation
0x80A9	DB length is exceeded with wrong offset or symbol size
0x80AE	DB changed during download in RUN after "Init" operation.
0x80C4	Lifetime buffer is corrupted. Reconnect again.

Return values of OP_READ mode

8.5 "OP_WRITE" mode

8.5.1 Writing data from Linux applications

After successful connection and configuration, the CPU can start exchanging data.

Writing data

For writing data, use ODKP_ISC with "OP_WRITE" (ID: 2).

Preconditions

Before you call write operations, make sure that you have carried out the following steps:

1. You have called "OP_INIT" successfully.
   For more information on "OP_INIT", refer to section "OP_INIT" mode (Page 64).
2. You have called "OP_CONNECT" successfully.
   For more information on "OP_CONNECT", refer to section "OP_CONNECT" mode (Page 69).

Calling the write operation

The following example shows the parameters of the ODKP_ISC "Write" mode.

Code	Comments

```
#Result_Write := ODKP_ISC(OperationID := "OP_WRITE",
    ParPtr1 := "RIB".EnableWrite,                    // Enable write operation
    ParPtr2 := "S7_SWCPU_DB",                        // Provided DB
    ParPtr3 := NULL,
    ParPtr4 := NULL,
    ParPtr5 := NULL);
```

The following table gives an overview of the available OP_WRITE mode parameters and their characteristics.

Section	Name	Data type	Description
Automatically generated parameters by ODKP_ISC			
Output	STATUS	INT	Function result error message
User defined parameters			
Input	OperationID	INT	OP_WRITE (id: 2)
InOut	ParPtr1	BOOL	Enabler for executing WRITE operation
InOut	ParPtr2	Type DB	Provided DB
InOut	ParPtr3	NULL	reserved
InOut	ParPtr4	NULL	reserved
InOut	ParPtr5	NULL	reserved

Parameters of OP_WRITE mode

The OP_WRITE mode of ODKP_ISC supports a maximum of 16 DBs. The OP_WRITE mode can be called multiple times with different provided DBs.

The write operation only works with the enabler state being TRUE.

The same DB must be used for both "init" and "write" operations. Changes in the DB (adding or deleting symbols) is not allowed.

RIB V1 does not support download in RUN.

If you use a cyclic OB for a write operation, set the OB priority to 16 or higher.

Return codes of STATUS variable

Calling the "write" operation might return the following codes in the STATUS variable.

STATUS	Reason
0x0000	Write operation successful
0x7000	No active job
0x8154	Invalid data type for ParPtr1
0x8254	Invalid data type for ParPtr2

STATUS	Reason
0x80A2	Internal error Contact customer support
0x80A4	Connect operation not finished yet
0x80A7	Provided DB not registered before with "Init" operation
0x80AE	DB changed during download in run after initialize operation
0x80C4	Lifetime buffer is corrupted. Reconnect again.

Return values of OP_WRITE mode

8.6 "OP_DISCONNECT" mode

8.6.1 Disconnecting from RIB

When you have finished exchanging data between the CPU and Linux applications, disconnect from RIB. We recommend that you disconnect from RIB before changing the CPU operating state to STOP or closing the RIB_App.

Disconnection

For disconnecting from RIB, use ODKP_ISC (SFC65490) with mode "OP_DISCONNECT" (ID: 4).

Calling the disconnect operation

The following example shows the parameters of the ODKP_ISC "Disconnect" mode.

Code	Comments
`#Result_Disconnect := ODKP_ISC(OperationID := "OP_DISCONNECT",`	
`ParPtr1 := "RIB".DisconnectRequest,`	`// Rising edge control for executing the operation`
`ParPtr2 := NULL,`	
`ParPtr3 := NULL,`	
`ParPtr4 := NULL,`	
`ParPtr5 := NULL);`	

The following table gives an overview of the available OP_DISCONNECT mode parameters and their characteristics.

Section	Name	Data type	Description
Automatically generated parameters by ODKP_ISC			
Output	STATUS	INT	Function result error message
User defined parameters			
Input	OperationID	INT	OP_DISCONNECT (id: 4)
InOut	ParPtr1	BOOL	Rising edge control variable to execute DISCONNECT operation
InOut	ParPtr2	NULL	reserved
InOut	ParPtr3	NULL	reserved
InOut	ParPtr4	NULL	reserved
InOut	ParPtr5	NULL	reserved

Parameters of OP_Disconnect mode

The "Disconnect" operation only works on the rising edge.

Return codes of STATUS variable

Calling the "Disconnect" operation might return the following codes in the STATUS variable:

STATUS	Reason
0x0000	Disconnection request successful
0x7000	No active job
0x7001	Disconnection in progress
0x7002	Disconnection in progress already
0x80A2	Internal error, contact customer support
0x8154	Invalid data type for ParPtr1
0x80E8	Connection in progress
0x80E9	Connection lost already
0x80EA	Disconnect operation executed after connection to RIB App lost

Return values of OP_DISCONNECT mode

Commissioning (software) 9

9.1 Cleaning lifetime buffer

The system cleans the lifetime buffer of an application when terminating this application. In specific cases, however, it is necessary to clean the lifetime buffer manually.

Cleaning the lifetime buffer area is necessary, if:

- An application has not been terminated properly
- An error occurs when closing the lifetime buffer
- The provider application cannot connect to RIB and the error "Shared memory segment is already open" appears. This error message might also indicate that a running application uses the same shared memory name.

Cleaning the entire lifetime buffer is only necessary in the following case:

- The memory of the lifetime buffer is corrupted

Note

Deleting the lifetime buffer on Linux-only systems and the CPU

On Linux-only systems, the lifetime buffers are stored in the Linux file system. Delete the shared memory by deleting the corresponding file from `/dev/shm`.

On the CPU, the lifetime buffers are stored in the Hypervisor shared memory area and can only be accessed with the corresponding driver.

Requirements

To clean the entire lifetime buffer, the following requirements apply:

- All applications using RIB are closed
- The operating state of the CPU is STOP or POWER OFF

To remove the lifetime buffer(s), the following requirements apply:

- All applications which access the lifetime buffer must be closed

s7_RIB_memory_cleaner

For cleaning the lifetime buffer, use the application s7_RIB_memory_cleaner. This application is installed together with the RIB_App.

The application has the following options:

- Displaying help messages
  For displaying help messages, use -h or--help.
  Usage: s7_RIB_memory_cleaner -h
- Cleaning the entire lifetime buffer
  For cleaning the entire lifetime buffer, a hard reset is required. For a hard reset, call the memory cleaner application with the force option (-f or --force).
  Usage: s7_RIB_memory_cleaner -f

  Note

  Software Controller

  Note that the s7_RIB_memory_cleaner may affect the stability of running applications. The tool is not intended for productive use. We recommend to remove the tool from a productive system.

  When carrying out this function, make sure that there is no running application using the RIB. The CPU must be in STOP or POWER OFF.

- Removing the lifetime buffer area by its name
  For removing the lifetime buffer area by its name, call the s7_RIB_memory_cleaner application with the name option (-n or --name). Use the name of the lifetime buffer as parameter.
  Usage: s7_RIB_memory_cleaner -n <LTB_Name>

  Note

  Software Controller and CPU 1518(F)-4 PN/DP MFP

  Note that the s7_RIB_memory_cleaner may affect the stability of running applications. The tool is not intended for productive use. We recommend to remove the tool from a productive system.

  When carrying out this function, make sure that no application is using the lifetime buffer and that there is no data being exchanged. Otherwise, removing the lifetime buffer could cause the application to crash or to return unexpected error codes.

- Removing all available lifetime buffer areas
  For removing all available lifetime buffer areas, call the s7_RIB_memory_cleaner application with the all option (-a or --all).
  Usage: s7_RIB_memory_cleaner -a
- Listing the names of all available lifetime buffer areas
  For listing all lifetime buffer areas, call the s7_RIB_memory_cleaner application with the list option (-l or --list).

Note

CPU 1518(F)-4 PN/DP MFP does not support -l and -a switches

Note that the CPU 1518(F)-4 PN/DP MFP does not support the -l and -a switches of the s7_RIB_memory_cleaner tool.

If you want to use the -f switch, the CPU user program must be in STOP.

Examples and tables A

A.1 Example project

Under <mount_point>/SWCPU/RIB you will also find a TIA Portal example project and an example C++ application.

You can use the example to adapt the values to your actual project.

To use the TIA Portal example project, copy the file "RIB_Template_SWCPU_DataExchange.ap18" to your Windows environment and import the file into TIA Portal.

Note

Example project is based on fail-safe CPU

The TIA Portal example project is based on a fail-safe CPU. If you are using a standard CPU, make sure to change the CPU type before downloading the project to the CPU. A fail-safe CPU cannot be downloaded to a standard CPU.

Structure of the example project

The example project has the following structure:

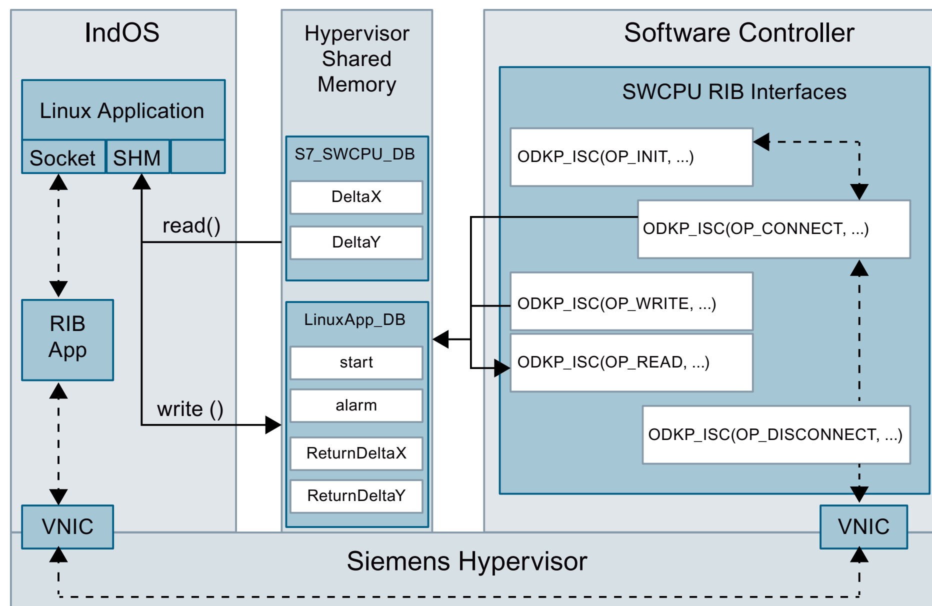


Figure A-1 Structure of example project

In the example project, the Linux Application provides four symbols. However, the Software Controller (SWCPU) only consumes three of them (start, ReturnDeltaX and ReturnDeltaY). The user data type (UDT) that stores these three symbols is T_LinuxApp_DB.

The Software Controller provides two symbols (DeltaX and DeltaY). The user data type that stores these two symbols is T_SWCPU_DB.

Creating user data types

The following image shows the already created user data in TIA Portal.

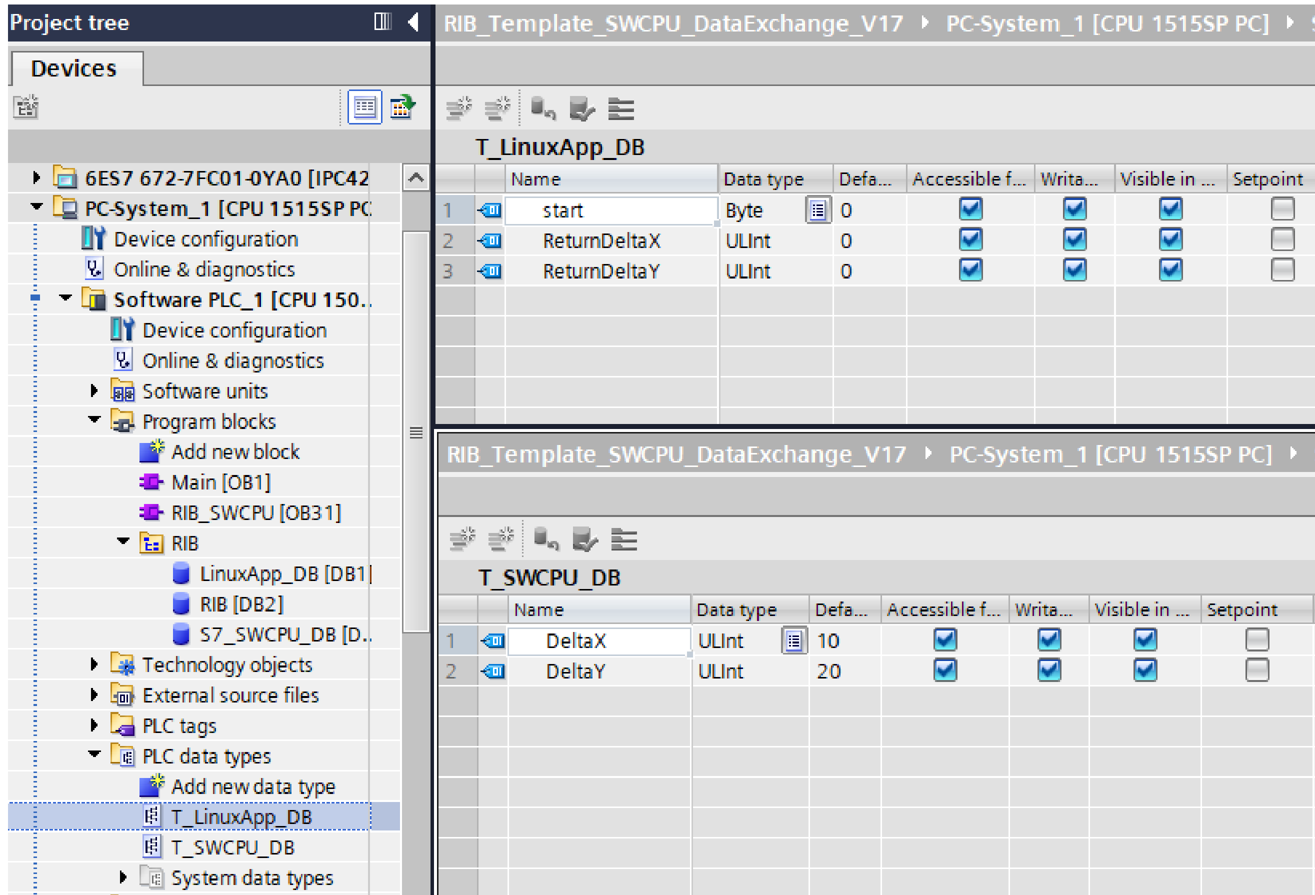


Figure A-2 Added user data types

Creating data blocks

The SFC will later be called with a data block (DB).

The following image shows how to create one of the two needed DBs with our defined user data types. In the example project, the data blocks S7_SWCPU_DB and LinuxApp_DB have already been created.

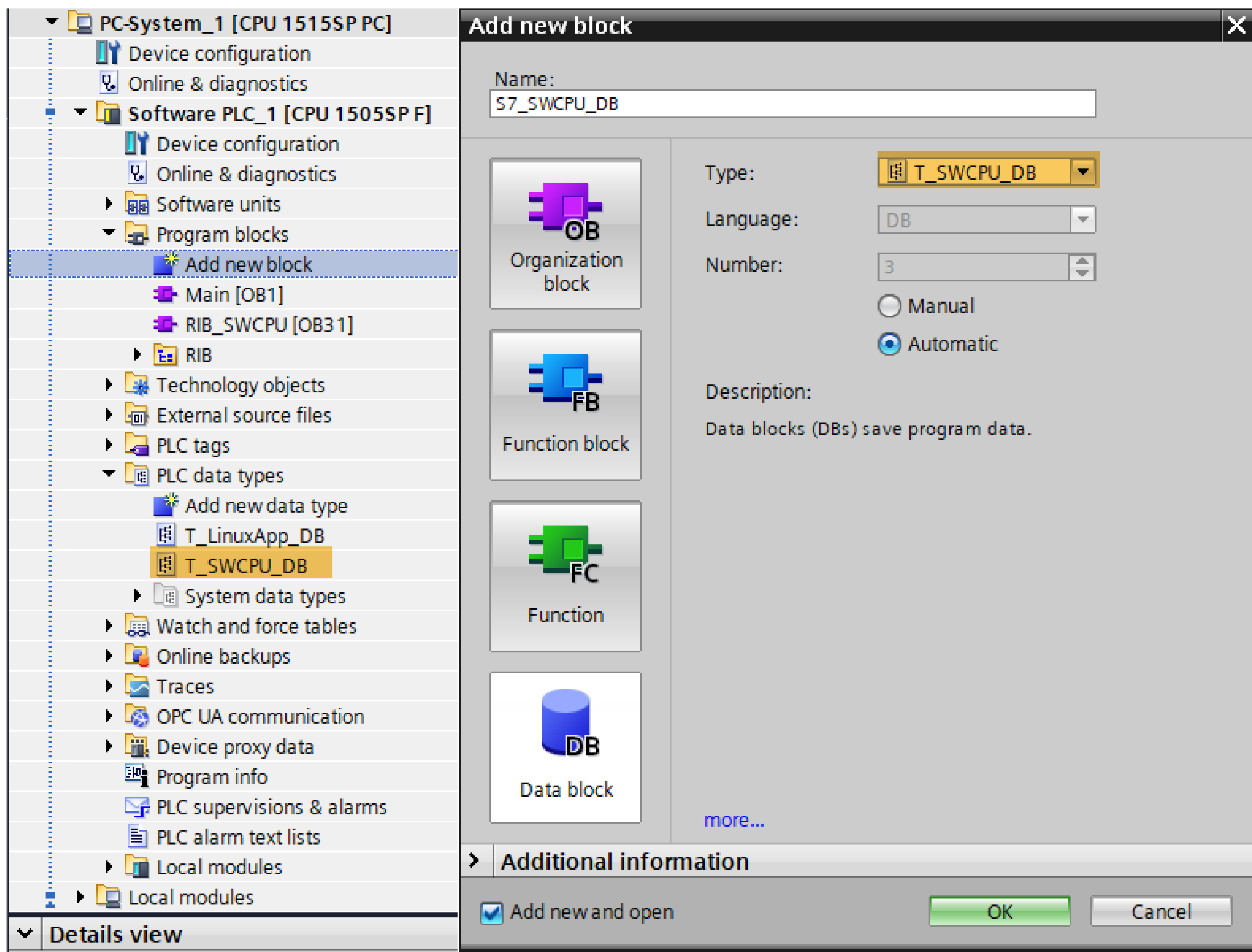


Figure A-3 Creating DBs

Calling SFC functions

The following image shows how to call SFC functions using the configured symbols:

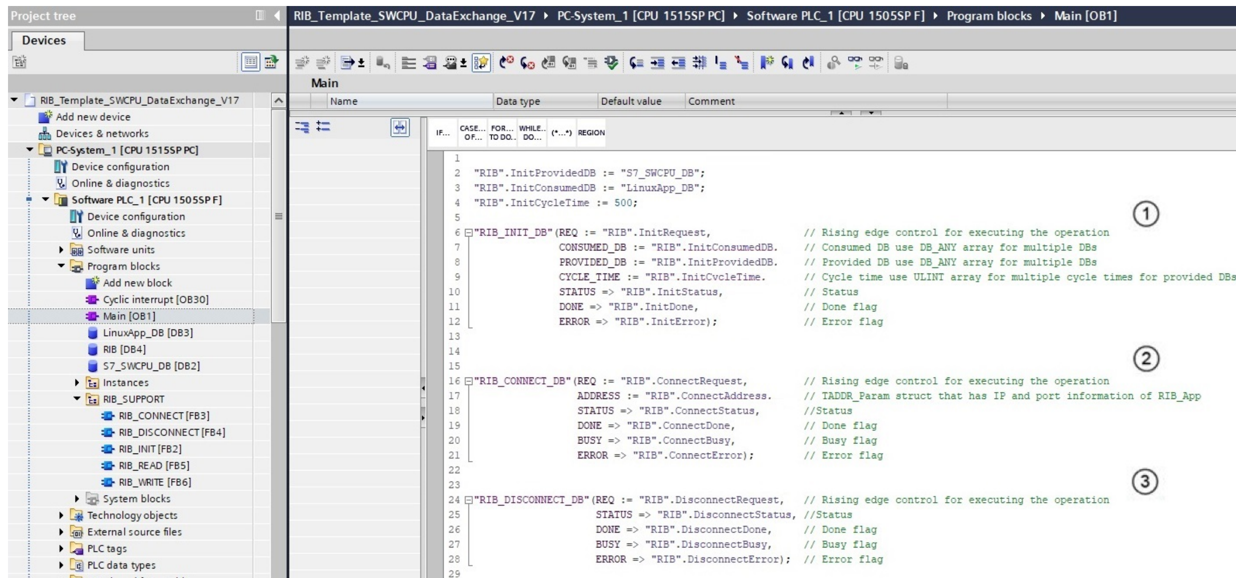


Figure A-4 SFC functions

In the example, we created wrapper FBs to make using the RIB functionality easy. We added DONE, ERROR and BUSY flags to facilitate programming. You can find the wrapper FBs in the project tree under "Programm blocks < RIB_SUPPORT".

For provided DBs and consumed DBs, we must create two variables (DB_ANY or DB_ANY array) in another DB. In the example, we used RIB[DB4] for storing variables for provided and consumed DBs.

Since we have one provided DB (S7_SWCPU_DB) and one consumed DB (LinuxApp_DB), we created two different DB_ANY variables ("RIB".ProvidedDB and "RIB".ConsumedDB).

For the provided DB, we assigned the cycle time in microseconds to an unsigned integer variable.

Note

Assignment operations in Main DB

Note that for display purposes, we put the assignment operations of this example in the Main OB. In practical use cases where assignment operations should not repeat every cycle, you can use a different OB.

The following table lists the commands and comments from section ① (code lines 6 to 12):

The RIB_INIT mode of ODKP_ISC is called with the consumed DB, provided DB and the cycle time of the provided DB		
Line	**Command**	**Comment**
6	`"RIB_INIT_DB" (REQ := "RIB".InitRequest,`	Rising edge control for executing the operation
7	`CONSUMED_DB := "RIB".InitConsumedDB,`	Consumed DB use DB_ANY array for multiple DBs
8	`PROVIDED_DB := "RIB".InitProvidedDB,`	Provided DB use DB_ANY array for multiple DBs

The RIB_INIT mode of ODKP_ISC is called with the consumed DB, provided DB and the cycle time of the provided DB		
9	`CYCLE_TIME := "RIB".InitCycleTime,`	Cycle time use ULINT array for multiple cycle times for provided DBs
10	`STATUS => "RIB".InitStatus,`	Status
11	`DONE => "RIB".InitDone,`	Done flag
12	`ERROR => "RIB".InitError);`	Error flag

The following table lists the commands and comments from section ② (code lines 16 to 21):

The RIB_CONNECT mode of ODKP_ISC is called with the Linux VNIC IP address and the port number of RIB_App		
Line	**Command**	**Comment**
16	`"RIB_CONNECT_DB" (REQ := "RIB".ConnectRequest,`	Rising edge control for executing the operation
17	`ADDRESS := "RIB".ConnectAddress,`	TADDR_Param struct that has IP and port information of RIB_App
18	`STATUS => "RIB".ConnectStatus,`	Status
19	`DONE => "RIB".ConnectDone,`	Done flag
20	`BUSY => "RIB".ConnectBusy,`	Busy flag
21	`ERROR => "RIB".ConnectError);`	Error flag

The following table lists the commands and comments from section ③ (code lines 24 to 28):

The RIB_DISCONNECT mode of ODKP_ISC must be called, if you want to disconnect and reconfigure the symbols or set the CPU to STOP		
Line	**Command**	**Comment**
24	`"RIB_DISCONNECT_DB" (REQ := "RIB".DisconnectRequest,`	Rising edge control for executing the operation
25	`STATUS => "RIB".DisconnectStatus,`	Status
26	`DONE => "RIB".DisconnectDone,`	Done flag
27	`BUSY => "RIB".DisconnectBusy,`	Busy flag
28	`ERROR => "RIB".DisconnectError);`	Error flag

Using SFC operations for exchanging data

The following image shows how to use SFC operations to exchange data between IndOS and the Software Controller:

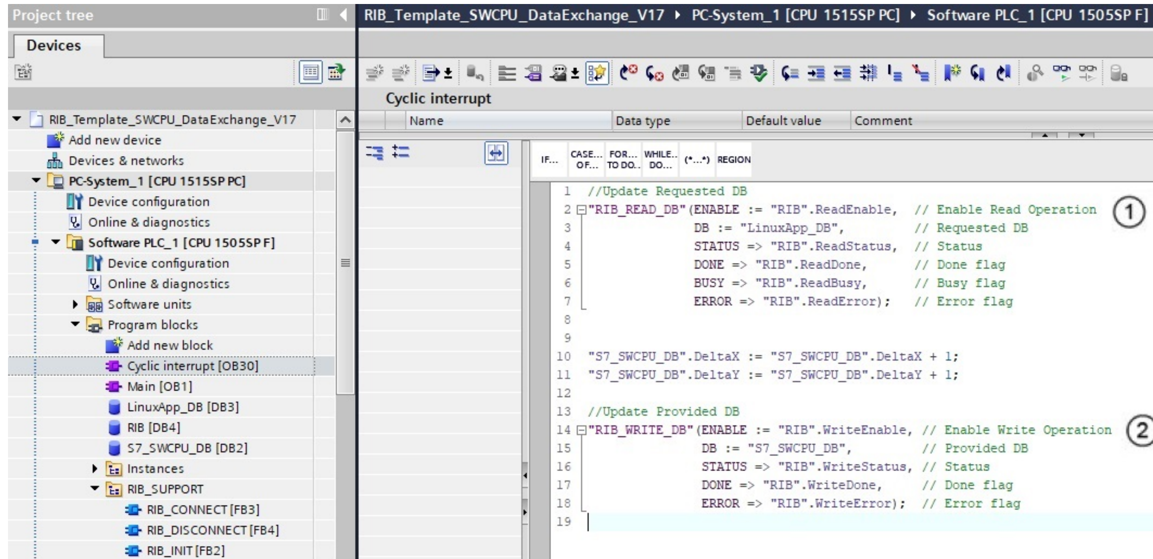


Figure A-5 SFC data exchange

The following table lists the commands and comments from section ① (code lines 2 to 7):

Line	Command	Comment
2	"RIB_READ_DB" (ENABLE := "RIB".ReadEnable,	Enable Read Operation (RIB_Read has a BUSY flag that is not available on RIB_WRITE. The BUSY flag is set to TRUE when requested symbols have not been shared by Linux applications.)
3	DB := "LinuxApp_DB",	Requested DB
4	STATUS => "RIB".ReadStatus	Status
5	DONE => "RIB".ReadDone,	Done flag
6	BUSY => "RIB".ReadBusy,	Busy flag
7	ERROR => "RIB".ReadError);	Error flag

The following table lists the commands and comments from section ② (code lines 14 to 18):

Line	Command	Comment
14	"RIB_WRITE_DB" (ENABLE := "RIB".WriteEnable,	Enable Write Operation (RIB_WRITE mode is called with the provided DB, in this example project, the provided DB is S7_SWCPU_DB)
15	DB := "S7_SWCPU_DB",	Provided DB
16	STATUS => "RIB":WriteStatus,	Status
17	DONE => "RIB".WriteDone,	Done flag
18	ERROR => "RIB".WriteError);	Error flag

Functional principle of using multiple DBs

In the above shown example project, we created the following constellation (Constellation I):

DB	Name
S7_SWCPU_DB	ProvidedDB
LinuxApp_DB	ConsumedDB

The following examples show the correct variables for the following constellation (Constellation II):

DB	Name
S7_SWCPU_DB S7_SWCPU_DB_2	ProvidedDB[0] ProvidedDB[1]
LinuxApp_DB LinuxApp_DB_2	ConsumedDB[0] ConsumedDB[1]

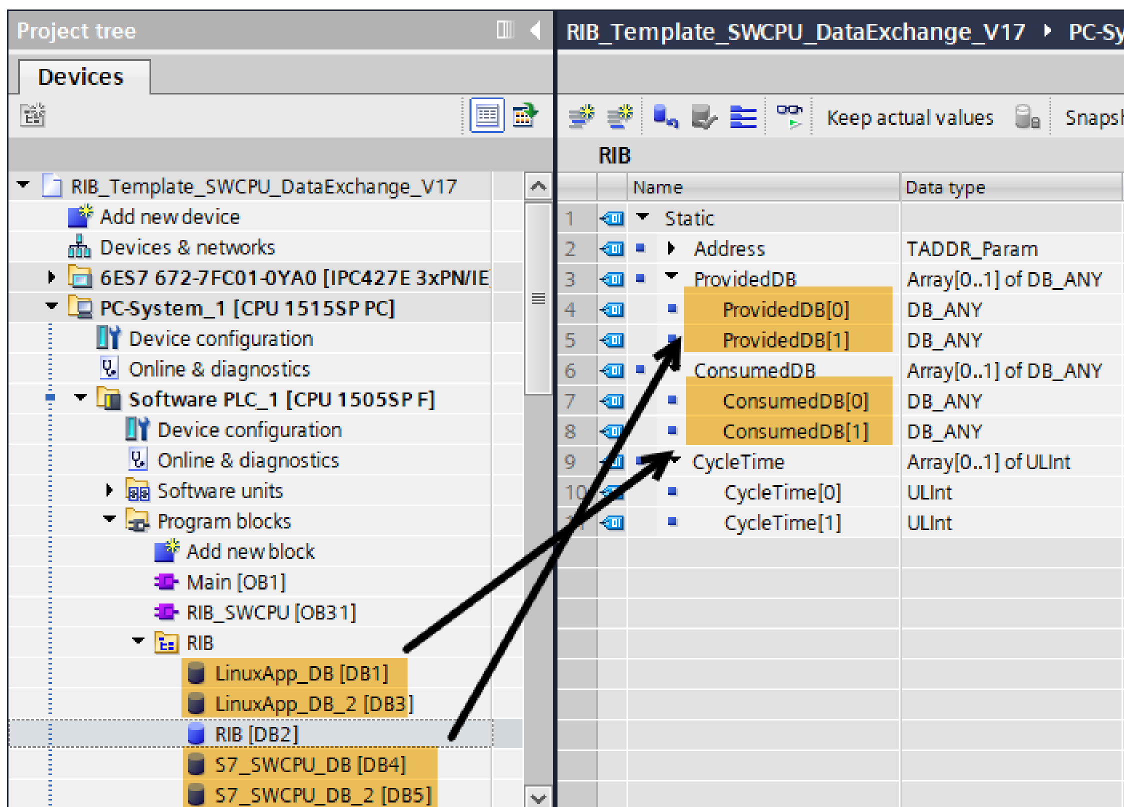


Figure A-6 Configuration of Constellation II

Instead of creating the variable as DB_ANY, we created a DB_Any array. One of the arrays stores the information of the consumed DBs and the other array the information of the provided DBs.

Since there are two provided DBs, we changed the data type to ULInt array CycleTime variable.

The following image shows the SFC operation with multiple DBs.

```
1
2  "RIB".InitConsumedDB[0] := "LinuxApp_DB";        ①
3  "RIB".InitConsumedDB[1] := "LinuxApp_DB_2";
4
5  "RIB".InitProvidedDB[0] := "S7_SWCPU_DB";        ②
6  "RIB".InitCycleTime[0] := 500;
7
8  "RIB".InitProvidedDB[1] := "S7_SWCPU_DB_2";      ③
9  "RIB".InitCycleTime[1] := 2500;
10
11 "RIB_INIT_DB"(REQ := "RIB".InitRequest,            // Rising edge control for executing the operation
12               CONSUMED_DB := "RIB".InitConsumedDB,  // Consumed DB use DB_ANY array for multiple DBs
13               PROVIDED_DB := "RIB".InitProvidedDB,  // Provided DB use DB_ANY array for multiple DBs
14               CYCLE_TIME := "RIB".InitCycleTime,    // Cycle time use ULINT array for multiple cycle time for provided DBs
15               STATUS => "RIB".InitStatus,           // Status
16               DONE => "RIB".InitDone,               // Done flag
17               ERROR => "RIB".InitError);            // Error flag
18
```

① Both consumed DBs are assigned to the consumed DB_ANY array.

② One of the provided DBs is assigned to the provided DB_ANY array as first element. The cycle time of the provided DB is assigned to the cycle time array with the corresponding index. The index of the cycle time and the provided DB array must match.

③ The other provided DB is assigned to the DB_ANY array as second element. The cycle time of the provided DB is also assigned to the matching index. If both provided DBs have the same cycle time, then, instead of the cycle time array, only one value can be used. This default cycle time value will be applicable to both provided DBs.

Figure A-7 Multiple DBs

Note

Multiple calls of read and write operations

Read and write operations can be called multiple times with different provided/consumed DBs but cannot be called at the same time.

A.2 Application examples

The following example project "SWCPU_DataExchange.cpp" relates to the TIA Portal example project and is also included the template applications folder "/home[/user]/rib".

```
//////////////////////////////////////////////////////////////////////
//////////////////////
/// Example app which communicates with PLC
///
/// \file SWCPU_DataExchange.cpp
///
///
**********************************************************************
******************/
/// \copyright Copyright (C) 2022 Siemens Aktiengesellschaft. All
rights reserved.
///
**********************************************************************
******************/
/// This program is protected by German copyright law and
```

```
international
/// treaties. The use of this software including but not limited to
its
/// Source Code is subject to restrictions as agreed in the license
/// agreement between you and Siemens.
/// Copying or distribution is not allowed unless expressly permitted
/// according to your license agreement with Siemens.
///
////////////////////////////////////////////////////////////////////
//////////////////////

#include "ribClient.h"
#include "symbolDescription.h"
#include <csignal>
#include <unistd.h>
#include <iostream>
#include <memory>
#include <cstdint>

using namespace RIB;

volatile bool stopProcess = false;

void ctrl_c_handler(int signal);

void ctrl_c_handler(int signal)
{
    (void)signal;
    std::cout << "Caught Stop-signal" << std::endl;
    stopProcess = true;
}

///
/// This app demostrates the feasibility of the lock free buffer
concept for RIB data exchange
///
int main(void)
{
    const std::uint64_t cycleTimeIn_usec = 150;
    const std::string ipAddr = "127.0.0.1";

    // register signal handler function
    signal(SIGINT, ctrl_c_handler);


    //*************************
    //Create connection object, specify request data and connect to
RIB
    RibClient ribClient = RIB::RibClient();

    //*************************
```

```
    // Create a memory area called "SWCPU_DataExchange" for
provided symbols
    const std::string providedMemoryName = "SWCPU_DataExchange";

    std::list<RIB::SymbolDescription> providedData
    {
        RIB::SymbolDescription("start",
RIB::SymbolDescription::DataType::UINT8_T),
        RIB::SymbolDescription("alarm",
RIB::SymbolDescription::DataType::UINT64_T),
        RIB::SymbolDescription("ReturnDeltaX",
RIB::SymbolDescription::DataType::UINT64_T),
        RIB::SymbolDescription("ReturnDeltaY",
RIB::SymbolDescription::DataType::UINT64_T)
    };

    ribClient.initProvidedData(providedMemoryName, providedData,
cycleTimeIn_usec);

    //**************************
    // Add requested symbols provided by SWCPU
    std::list<std::string> consumedData
    {
        "DeltaX",
        "DeltaY"
    };

    ribClient.initConsumedData(consumedData);


    if (ribClient.activate() != RibReturnCode::OK)
    {
        std::cout << "Activating RIB was not possible" << std::endl;
        ribClient.deactivate();
        return EXIT_FAILURE;
    }

    // get the writer for this client to write the data
    auto [retVal1, writer] =
ribClient.getWriter(providedMemoryName);
    if (retVal1 != RIB::RibReturnCode::OK)
    {
        std::cout << "Getting the writer for this client was not
possible" << std::endl;
        ribClient.deactivate();
        return EXIT_FAILURE;
    }

    // get the reader for this client to read the data
    RibReturnCode getReaderResult;
    std::shared_ptr<RIB::Reader> reader = nullptr;
    std::tie(getReaderResult, reader) = ribClient.getReader();
```

```
    if (getReaderResult != RIB::RibReturnCode::OK)
    {
        std::cout << "Getting the reader for this client was not 
possible." << std::endl;
        ribClient.deactivate();
        return EXIT_FAILURE;
    }

    // define pointers to read the payload.
    uint64_t* deltaX = nullptr;
    uint64_t* deltaY = nullptr;

    // get a pointer to access the requested data in the symbol 
image and cast it to specific type
    auto providedSymbolNameToPointerMap = writer-
>getSymbolNameToPointerMap();
    auto returnDeltaX = 
static_cast<uint64_t*>(providedSymbolNameToPointerMap["ReturnDeltaX"
]);
    auto returnDeltaY = 
static_cast<uint64_t*>(providedSymbolNameToPointerMap["ReturnDeltaY"
]);

    while (!stopProcess)
    {
        //Update requested symbol values
        auto [retValRead, updateAvail] = reader->read();
        if (retValRead != RibReturnCode::OK &&
           retValRead != RibReturnCode::BufferNotWrittenByProducer 
&&
           retValRead != RibReturnCode::DataNotAvailable)
        {
            // react on error
        }

        if (updateAvail)
        {
            // there is a change in the availability of symbols, so 
ask for update
            auto symbolNameToPointerMap = reader-
>getSymbolNameToPointerMap();
            deltaX = 
static_cast<uint64_t*>(symbolNameToPointerMap["DeltaX"]);
            deltaY = 
static_cast<uint64_t*>(symbolNameToPointerMap["DeltaY"]);
        }

        //Consumed DeltaX and DeltaY symbols are mirrored on 
providedData;
        if(deltaX != nullptr && deltaY != nullptr)
        {
```

```
            *returnDeltaX = *deltaX;
            *returnDeltaY = *deltaY;
        }

        // Update provided symbol values
        auto retVal = writer->write();
        if (retVal != RIB::RibReturnCode::OK)
        {
            std::cout << "Writing data caused an error" <<
std::endl;
            ribClient.deactivate();
            return EXIT_FAILURE;
        }

        usleep(cycleTimeIn_usec);
    }

    if (ribClient.deactivate() != RIB::RibReturnCode::OK)
    {
        std::cout << "Error: Deactivation of RibClient caused an
Error" << std::endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

This example provides a rough overview of the application flow. Error handling is omitted to obtain a better overview.

Note

Note that the CPPYY package should be installed for the Python examples.

```
#include "RibClient.h"

int main()
{
  // 1. Initialize the RibClient object
  RIB::RibClient ribClient;


  // 2. Configure RIB consumer part
  ribClient.initConsumedData("hugo");
  ribClient.initConsumedData("temperature_1");


  // 3. Configure RIB provider part
  string shmName = "myOwnShm";
  int cycleTimeInMilliseconds = 500;
  const list<RIB::SymbolDescription> symbolList = {
```

```
    {"myValue_x", RIB::SymbolDescription::DataType::UINT64_T, 1},
    {"myValue_x", RIB::SymbolDescription::DataType::UINT64_T, 1},
  ribClient.initProvidedData(shmName, symbolList,
cycleTimeInMilliseconds);

  // 4. Connect to RIB App, create SHM, register symbols, create
symbol images of provided
  // and consumed data, start connection thread
  ribClient.activate();

  // 5. Use symbols
  // 5.1 Access writer
  auto [getWriterResult, writer] = ribClient.getWriter(shmName);

  // 5.2 Access pointer to symbols to write
  auto [getPointerResultX, ptrToMyValueX] = writer-
>getPointerToSymbol<std::uint64_t>("myValue_x");
  auto [getPointerResultY, ptrToMyValueY] = writer-
>getPointerToSymbol<std::uint64_t>("myValue_y");

  // 5.3 Access reader
  auto [getReaderResult, reader] = ribClient.getReader();

  // 5.4 Access pointer to 'hugo' and 'temperature_1'.
  auto [getPointerResult, ptrToSymbolHugo] = reader-
>getPointerToSymbol<std::uint64_t>("hugo");
  auto [getPointerResult, ptrToSymbolTemperature_1] = reader-
>getPointerToSymbol<std::uint64_t>("temperature_1");

  while (...)
  {
    // 5.5 Read data
    reader->read();

    // 5.6 User algorithm
    *ptrToMyValueX = *ptrToSymbolHugo + 1;
    *ptrToMyValueX = *ptrToSymbolTemperature_1 * 42;

    // 5.7 Write data
    writer->write();
  }

  //6. Cleanup RIB
  ribClient.deactivate();

  return 0;
}
```

A.3 Allowed data types

Currently, the following data types and arrays of them are allowed for the definition of symbols:

Category	Data types	Size in bytes
Signed integer	int8_t, int16_t, int32_t, int64_t	1, 2, 4, 8
Unsigned integer	uint8_t, uint16_t, uint32_t, uint64_t	1, 2, 4, 8
Floating points	float, double	4, 8

Use these type names for the place holder "<SymbolType>" in the subsequent configuration descriptions.

A.4 Example configuration string

The following example shows a configuration JSON string and descriptive comments.

Code	Comment
`"Type": "ConnectToRIBConfig",`	`// mandatory, this string represents configuration data to connect an application to the RIB`
`"Version": "1.0",`	`// mandatory, the version of the connection data information (needed for compatibility issues)`
`"<Application_Name>":`	`// mandatory, all information needed to register an application to the RIB, this field represents the name of the application`
`{`	
`    "Type": "ApplicationData",`	`// mandatory, the type that is described here is an ApplicationData object`
`    "PID": <PIDOfApplication>,`	`// mandatory, the current process ID of the application`
`    "Description": "<DescriptionOfApp>",`	`// optional, some more information regarding the application`
`    "Version": "<VersionOfApp>",`	`// optional, the version of the application`
`    "Manufacturer": "<ManufacturerOfApp>",`	`// optional, the creator of the application. e.g. company name`
`    "Provides":`	`// optional, if the application provides data, this data is described here`
`    {`	
`        "<IdentifierOfTheSharedMem>":`	`// mandatory, if the application provides data, the ID of the shared memory is needed`
`        {`	
`            "Type": "Provide";`	`// mandatory, the type that is described here is a Provide object`
`            "Signal": <SignalId>,`	`// mandatory, the signal the client must listen to (currently not implemented, -1)`

```
            "CycleTimeInMicroseconds":        // optional, the cycleTime the buffer is
<CycleTime>,                                  updated in microseconds, can be added
                                              optionally
            "Version": "<VersionOfApp>",      // optional, the version of the data set can
                                              be entered here
            "Symbols":                        // mandatory, at least one data symbol must
                                              be provided
            {
                "Symbol_1" :                  // mandatory, name of the Symbol
                {
                  "Offset": "<AddressOffset>", // mandatory, offset of the symbol in the
                                              shared memory in bytes
                    "Size": "<SizesInBytes>", // mandatory, size of the symbol in the
                                              shared memory in bytes
                    "Type": "<SymbolType>"
                },
                ...
                "Symbol_n" :                  // optional, another symbol
                {
                  "Offset": "<AddressOffset>", // mandatory, offset of the symbol in the
                                              shared memory in bytes
                    "Size": "<SizesInBytes>", // mandatory, size of the symbol in the
                                              shared memory in bytes
                    "Type": "<SymbolType>"    // mandatory, the type of the symbol (in
                                              first version only simple value types are
                                              allowed)
                }
            }
        }
    },
    "Requests":                               // optional, if the application requests
                                              data, this data is described here
    {
        "Symbols":                            // mandatory, at least one data symbol must
                                              be requested
        [
            "Symbol_1",                       // mandatory, name of the Symbol
            ...
            "Symbol_n"                        // optional, another symbol name
        ]
    }
}
```

A.5 RibReturnCode

The following table lists all API return codes and their meanings.

Value	Error	Description
0	`OK`	*Operation was successful.*
100	`NotConnected`	*Connecting to RIB failed, the client could not establish a socket connection to RIB.*
101	`NotSignedIn`	*Connecting to RIB failed, the client configuration was not accepted by RIB.*
102	`NotSignedInInvalidJson`	*Connecting to RIB failed, the client configuration was not accepted by RIB, invalid JSON format.*
103	`NotSignedInAppAlreadyExists`	*Connecting to RIB failed, the client configuration was not accepted by RIB, application with same name already exists.*
104	`NotSignedInProvidedSymbolAlreadyExists`	*Connecting to RIB failed, the client configuration was not accepted by RIB, one of the provided symbols already exists in RIB.*
105	`NotSignedInProvidedSymbolInvalidType`	*Connecting to the RIB failed, the client configuration was not accepted by RIB, the type of a provided symbol is not supported.*
106	`SocketCommunicationError`	*Operation failed because of an error while communicating with the RIB app over a socket.*
107	`RibEnvironmentConfigNotAvailable`	*Generating a lifetime buffer failed because RibEnvironmentConfig is not available.* (for example missing maximum guaranteed lifetime of buffer elements in lifetime buffer)
108	`GenerateLifetimeBufferFailed`	*Generating a lifetime buffer failed because internal allocation failed.*
109	`AddConfigurationError`	*Adding a configuration failed because client is already connected to RIB.*
110	`InvalidIPAddress`	*The IPAddress is not valid.*
111	`InvalidConfigurationData`	*ConfigurationData is not valid.*
112	`OperationNotAllowedWhenConnected`	*Operation is not allowed when application is already connected.*
113	`OperationNotAllowedWhenSignedIn`	*Operation is not allowed when application is already signed in.*
114	`AlreadySignedIn`	*Already signed in.*
200	`SignOutTimeOut`	*Disconnect from RIB failed, at least one consumer did not respond to RIB within the given timeout and might still access the lifetime buffer.*
201	`SignOutUnknownError`	*Disconnect from RIB failed, an unknown error happened and there might still be some client accessing the lifetime buffer.*
300	`WriteSymbolsError`	*Writing symbol data to shared memory failed.*
301	`WriteSymbolsInvalidParameter`	*Writing symbol data to shared memory failed, invalid parameter for data to write.*
302	`WriteSymbolsErrorInvalidSize`	*Writing symbol data to shared memory failed, invalid size for data to write.*
303	`AddingSymbolNameFailed`	*Adding a symbol name failed, symbol name already exists.*
400	`ReadTimeOut`	*Reading data failed because read time exceeded. Data may not be consistent.*
401	`InvalidBufferElement`	*Reading data failed because the last valid buffer element in the lifetime buffer is out of bounds.*
402	`BufferNotWrittenByProducer`	*Reading data failed because lifetime buffer has not been written yet by the producer.*

Value	Error	Description
403	DataNotAvailable	*Reading data failed because lifetime buffer is not available anymore, e.g. it was closed by the producer that disconnected from RIB.*
404	SharedMemoryNotAvailable	*Shared memory not found.*
405	ReadError	*Accessing the reader failed, the lifetime buffer is not available for reading.*
406	SymbolNotFound	*Symbol not found.*
407	InvalidBufferType	*Invalid buffer type*
408	InvalidBufferVersion	*Invalid buffer version*
600	InvalidVersion	*Version of RIB_App and library does not match.*
601	MessageTooLong	*The message which is tried to be sent is too long*

B List of abbreviations

The following list explains the abbreviations used in this manual:

API	Application Programming Interface
CPPYY	Automatic Python-C++ bindings
DB	Data Block
IndOS	SIMATIC Industrial OS
IPC	Industrial PC
JSON	JavaScript Object Notation
LTB	Lifetime Buffer
MFP	Multi Functional Platform
PID	Process Identification Number
POSIX	Portable Operating System Interface
RIB	Realtime Information Backbone
SHM	Shared Memory
SWCPU	Software Controller
TCP	Transmission Control Protocol
UDT	User Data Type
VMM	Virtual Machine Manager
VNIC	Virtual Network Interface Card