RTI Connext TSS Documentation

Release 4.2.0

Real-Time Innovations, Inc.

August 2026

Contents

RTI® Connext® TSS is a Future Airborne Capability Environment (FACE™) Transport Services Segment (TSS) powered by *RTI Connext*.

1 Introduction

1.1 What is FACE TSS?

FACE™ (Future Airborne Capability Environment) is an open software technical standard designed by the Open Group FACE Consortium to improve affordability, accelerate development time, and increase compatibility across different computing environments. This standard establishes a common operating environment, enabling software components to be reused across avionics systems that use the FACE Reference Architecture.

Within the FACE architecture, the TSS (Transport Services Segment) defines how data moves within a system to optimize component portability and streamline system development.

For detailed information, the FACE Technical Standard, Edition 3.2, is available for download from the FACE Open Group Library.

1.2 What is RTI Connext TSS?

RTI® Connext® TSS, part of the *RTI Connext* product suite, is a customizable TSS solution based upon the open Data Distribution Service (DDS™) standard. Conformant with FACE 3.2, *Connext TSS* provides the APIs and capabilities that FACE portable components use to share data. This data can be exchanged across multiple networks, safety levels, and components from different suppliers.

Connext TSS also provides a TPM (Transport Protocol Module) interface, allowing you to use different transports within the TSS. This feature lets you develop custom TPMs and integrate third-party TPMs with *Connext TSS* and the *Connext Professional* or *Connext Micro* TPM adapters.

For implementation guidance, see *Integrating a Compatible Third-Party TPM*.

This manual includes detailed information about installing, configuring, and using *Connext TSS*.

1.3 What are RTI Connext Professional and RTI Connext Micro?

RTI Connext Professional is a software framework for building connected and distributed applications with requirements for high performance and scalability. It implements the DDS standard in its software development kit (SDK) and a variety of tools and services. *RTI Connext Micro* provides a smaller, modular connectivity framework for resource-limited devices. Both *Connext Professional* and *Connext Micro* provide communications services for distributing time-critical data.

This manual may suggest reading sections of the *Connext Professional* or *Connext Micro* documentation. You can find these documents in your *Connext Professional* or *Connext Micro* installation directory, respectively.

Note: *Connext TSS* is compatible with both *Connext Professional* and *Connext Micro*. However, *Connext TSS* only supports the C API for *Connext Professional*. It supports both the C and C++ APIs for *Connext Micro*. For more information about compatibility, refer to the *Release Notes* in this manual.

1.4 About this Document

This manual is written for developers and engineers with a background in software development.

1.4.1 Paths Mentioned in Documentation

This documentation refers to:

- <RTITSSHOME> refers to the installation directory for *Connext TSS*.

  You may also see `$RTITSSHOME` on Linux or `%RTITSSHOME%` on Windows, which refer to an environment variable set to the installation path.

- <NDDSHOME> refers to the installation directory for *Connext Professional*.

  You may also see `$NDDSHOME` on Linux or `%NDDSHOME%` on Windows, which refer to an environment variable set to the installation path.

- <RTIMEHOME> refers to the installation directory for *Connext Micro*.

  You may also see `$RTIMEHOME` on Linux or `%RTIMEHOME%` on Windows, which refer to an environment variable set to the installation path.

- <path to examples> refers to the examples folder in the installation directory for *Connext TSS*.

Whenever you see <RTITSSHOME>, <NDDSHOME>, or <RTIMEHOME> used in a path, replace it with your installation path.

1.5 Support Resources

Besides this documentation, RTI provides additional resources for help with *Connext TSS* and related DDS products.

1.5.1 Community Forum

RTI has created a Connext TSS Special Interest Group (SIG) on our community site for advice, suggestions, or support. Note that anything you post to this forum is open to the public; please take precaution about any information you share.

1.5.2 RTI Connext Documentation

Documentation for *Connext Professional* and *Connext Micro* is available in the `doc` folder of your *Connext Professional* or *Connext Micro* installation directory.

Documentation for other RTI products is available on the RTI Community Portal.

1.5.3 Training and Professional Services

If you're new to DDS, the following resources can help you learn about this middleware standard:

- Connext Developer Guide. A series of short, independent, hands-on training modules introduce you to basic DDS concepts.
- Connext AI Chatbot. Get real-time assistance and comprehensive information about Connext features and functionality. Get more information about the chatbot here.

You can also contact the RTI Professional Services team or your local RTI representative to learn from our experts.

1.5.4 Technical Support

Phone: (408) 990-7444 Email: support@rti.com Website: https://support.rti.com/

2 Installation

2.1 Installing Connext TSS

The *Connext TSS* package is provided in a zip archive, `rti_connext_tss_<version>.zip`. The package is available for download from the RTI Customer Portal; if you do not have access, contact RTI Customer Support.

To install *Connext TSS*, unzip the archive with the following command:

```
unzip rti_connext_tss_<version>.zip
```

The *Connext TSS* libraries are shipped as source code. After installing, you must build the libraries before using *Connext TSS* with a FACE application; refer to *Building Connext TSS* for more information.

Note: The root directory where the archive is extracted is the *Connext TSS* installation directory, which is referred to as `<RTITSSHOME>` in this documentation. See *Paths Mentioned in Documentation* for a complete list of paths noted in this manual.

2.2 Package Contents

Folder or File	Description
`bin/`	Includes *RTI Code Generator*, referred to as *rtiddsgen* in this documentation. Modified specifically for *Connext TSS*, this tool is used to generate TSS-specific header and plug-in source files.
`doc/`	Includes the complete *Connext TSS* user's manual and API documentation in html format.
`code/C/include/`	Header files for C implementation of *Connext TSS*.
`code/C/src/`	Source files for the C implementation of *Connext TSS*.
`code/C++/include/`	Header files for the C++ wrapper of the C implementation of *Connext TSS*.
`code/C++/src/`	Source files for the C++ wrapper of the C implementation of *Connext TSS*.
`examples/`	An example FACE application that makes use of the *Connext TSS* library.
`resource/`	Includes CMake and *rtiddsgen* support files.
`sbom/`	Software Bill of Materials (SBOM) that RTI uses in this *Connext TSS* release.
`CMakeLists.txt`	A CMake script for generating the build environment for *Connext TSS* libraries.
`README.html`	Links to the *Connext TSS* user's manual provided in the `doc/` folder.
`RTI_License_Agreement.pdf`	RTI's Software License Agreement (SLA).

3 Building Connext TSS

The *Connext TSS* libraries are shipped as source and must be built before use by a FACE application. This chapter describes how to build *Connext TSS* libraries for an architecture supported by RTI (refer to *Supported Platforms* for a list of platforms supported in this release). Note that the instructions in the following sections must be performed in order.

3.1 Prerequisites

- *Connext Micro* or *Connext Professional* must be installed (and built, if using *Connext Micro*) for the appropriate target architecture and FACE profile; check the *Release Notes* for compatibility information. For building instructions, refer to the `doc` folder in your *Connext Micro* or *Connext Professional* installation directory.
- Install a Java Runtime Environment (JRE). *RTI Code Generator* (*rtiddsgen*) requires a JRE to execute; RTI recommends AdoptOpenJDK version 17.0.12 (LTS) or later.

  Note: If you installed *Connext TSS* with *Connext Professional*, a JRE is already included in your *Connext Professional* installation. Before you build *Connext TSS*, make sure that the included JRE is version 17.0.12 (LTS) or later.

- Set the `JREHOME` environment variable to the location of your JRE (if your JRE is not installed in your application's path).
- Download and install CMake version 3.20 or above. *Connext TSS* provides and uses CMake scripts to generate the environment to build its libraries and example applications.

3.2 The Host and Target Environment

The following terminology is used to refer to the environment in which *Connext TSS* is built and run:

- The *host* is the machine that runs the software to compile and link *Connext TSS*.
- The *target* is the machine that runs *Connext TSS*.
- In many cases *Connext TSS* is built *and* run on the same machine. This is referred to as a *self-hosted environment*.

The *environment* is the collection of tools, OS, compiler, linker, hardware, etc., needed to build and run applications.

The following sections describe how to set up both types of environments for compiling and running *Connext TSS*.

3.2.1 The host environment

Before building the *Connext TSS* source code, you must set some environment variables to the installation paths for *Connext TSS* and *Connext Micro* or *Connext Professional*.

1. Make sure CMake is in the **PATH** environment variable.
2. Set the `RTITSSHOME` environment variable to point to your *Connext TSS* installation directory.

   For example, if *Connext TSS* is installed in `/home/user/rti/rti_connext_tss-<version>/`, set the `RTITSSHOME` environment variable with this command (in bash):

   ```
   export RTITSSHOME=/home/user/rti/rti_connext_tss-<version>
   ```

3. Set the `RTIMEHOME` or `NDDSHOME` environment variable to point to your *Connext Micro* or *Connext Professional* installation, respectively.

   For example, if *Connext Micro* is installed in `/opt/local/rti/rti_connext_dds_micro-<version>`, then set `RTIMEHOME` with this command (in bash):

   ```
   export RTIMEHOME=/opt/local/rti/rti_connext_dds_micro-<version>
   ```

   Note: For Pro TPM builds, `NDDSHOME` must point to a non-`FACE_GP` *Connext Professional* library.

Note: The *Connext Micro* libraries must be conformant to the FACE profile of the *Connext TSS* libraries being built. For help configuring and building *Connext Micro* for FACE, refer to the user's manual in the `doc` folder of your *Connext Micro* installation directory.

3.2.2 The target environment

Connext TSS requires a CMake architecture file configured for the target environment where it will run.

Note: For a list of demonstrated architectures for *Connext TSS*, refer to *Supported Platforms*.

The `RTITSSHOME/resource` folder includes CMake architecture files for the following platforms:

- `x64Linux6gcc13.3.0FACE_GP`
- `x64Linux6gcc13.3.0FACE_SB`
- `x64Win64VS2022FACE_GP`
- `x86Lynx1782024.09.0.APEXgcc11.3.0FACE_SB`
- `x64Vx23.09llvm16.0_rtpFACE_GP`

Set the environment variable `RTITSSARCH` to the selected architecture. For example:

```
export RTITSSARCH=x64Linux6gcc13.3.0FACE_GP
```

If compiling *Connext TSS* for the SafetyBase FACE Profile for *Connext Micro*, you may need to set the `RTIMEARCH` environment variable and append the `CERT` suffix to the architecture name. For example:

```
export RTITSSARCH=x64Linux6gcc13.3.0FACE_SB
export RTIMEARCH=x64Linux4gcc7.3.0CERT
```

If compiling *Connext TSS* for the General Purpose FACE Profile for *Connext Professional*, you may need to set the `NDDSARCH` environment variable and append the `FACE_GP` suffix to the architecture name. For example:

```
export RTITSSARCH=x64Linux6gcc13.3.0FACE_GP
export NDDSARCH=x64Linux4gcc7.3.0FACE_GP
```

Note: To use a different architecture from those listed above, refer to *Porting Connext TSS*.

3.3 Building the Libraries

After setting up your host and target environments, use CMake to create an out-of-source build of *Connext TSS*.

1. Create a build directory:

   ```
   cd ${RTITSSHOME}
   mkdir build
   ```

2. Run CMake with the required definitions, as shown in the following command:

   ```
   cd build
   cmake -DRTI_CONNEXT_TYPE=<micro|pro> -DRTI_TSS_ENABLE_FACE_COMPLIANCE=<your_face_
   ↪profile> ../
   ```

   The CMake definitions in the above command are required to successfully build *Connext TSS*.

   - `-DRTI_CONNEXT_TYPE` sets *Connext Micro* (`micro`) or *Connext Professional* (`pro`) as the RTI product being used.
   - `-DRTI_TSS_ENABLE_FACE_COMPLIANCE` sets the FACE OSS (Operating System Segment) profile *Connext TSS* will conform to:
     - `None` (**default**)
     - `GeneralPurpose`
     - `SafetyExtended`
     - `SafetyBase`
     - `Security`

     Note: While `RTI_TSS_ENABLE_FACE_COMPLIANCE` can be set to any FACE OSS profile, check the *Supported Platforms* for information on which profiles are supported per platform. For more information about FACE profiles, see the FACE Technical Standard.

3. Build the *Connext TSS* libraries:

   ```
   cmake --build .
   ```

Alternatively, you can use the CMake GUI (`cmake-gui`) instead of the command line to complete the build.

3.3.1 Optional CMake definitions

The following definitions can be used to customize your build.

Definition	Description	Options
`-DCMAKE_BUILD_TYPE`	Sets the type of library or executable built.	• "*Release*" (**default**) • "*Debug*"
`-DBUILD_TSS`	Enables or disables building the Transport Service/Type Abstraction library.	• "*ON*" (**default**) • "*OFF*"
`-DBUILD_STRINGS_SEQUENCES`	Enables or disables building the RTI strings and sequences library.	• "*ON*" (**default**) • "*OFF*"
`-DBUILD_SERIALIZATION`[1]	Enables or disables building the FACE-to-CDR (Common Data Representation) serialization library.	• "*ON*" (**default**) • "*OFF*"
`-DBUILD_MICRO_TPM`	Enables or disables building the *Connext Micro* TPM library.	• "*ON*" (**default**) • "*OFF*"
`-DBUILD_PRO_TPM`	Enables or disables building the *Connext Professional* TPM library.	• "*ON*" • "*OFF*" (**default**)
`-DLANGUAGE`[2][3]	Sets the language to build the enabled libraries for C or C++.	• "*c*" (**default**) • "*cpp*"
`BUILD_SHARED_LIBS`[4]	Defines whether libraries are built as dynamic (on) or static (off).	• "*ON*" • "*OFF*" (**default**)
`-DLIFECYCLE_CLASS_EXT`	Enables or disables sequence element lifecycle operations for complex types. See *Sequence Element Lifecycle Operations*.	• "*ON*" (**default**) • "*OFF*"

3.3.2 Compile-time configuration

Connext TSS provides a set of compile-time parameters that control resource limits and buffer sizes. These parameters are defined with default values in `code/RTI_Connext_TSS_Config.h`. Each parameter is wrapped in an `#ifndef` guard, so it can be overridden by passing `-D<PARAMETER>=<value>` to CMake.

For example, to set the maximum number of connections per Base instance to 100:

```
cmake -DMAX_CONNECTIONS_PER_BASE_INSTANCE=100 ...
```

The following parameters can be overridden through CMake:

[1] Serialization libraries are only used with *Connext Micro*. CMake will not build these libraries if `-DBUILD_PRO_TPM` is enabled.
[2] RTI's C++ API is a wrapper around the C API. When C++ is selected, both C and C++ libraries are built.
[3] *Connext TSS* does not support C++ libraries on *Connext Professional*.
[4] This *Connext TSS* release does not support dynamic libraries on Windows systems or for SafetyBase profiles.

Parameter	Description	Default
STRINGS_BOUND	Default bound for unbounded strings.	255
SEQUENCES_BOUND	Default bound for unbounded sequences.	4096
MAX_CONNECTIONS_PER_BASE_INSTANCE	Maximum number of connections that a Base instance can own.	10000
MAX_TPM_REFERENCES_PER_BASE_INSTANCE	Maximum number of TPM references that a Base instance can hold.	50
MAX_CHANNELS_PER_CONNECTION	Maximum number of channels that a connection can contain.	10000
MAX_CONFIGURATION_STRING_LENGTH	Maximum length of the TSS configuration string. Update based on the size of your configuration.	1024
LOG_FILE_NAME_MAX_LENGTH	Maximum length of the log file name when logging is enabled.	256

Note: When using Connext Micro, data larger than the transport MTU is only fragmented for BEST_EFFORT reliability. RELIABLE communication does not support fragmentation and over-MTU samples are discarded by Micro. For this reason, keep effective serialized sample sizes within MTU by using bounded types and appropriately sizing STRINGS_BOUND and SEQUENCES_BOUND.

The following additional parameters are defined in RTI_Connext_TSS_Config.h but are not passed through CMake. To override them, define them in your compiler flags or edit the header directly:

Parameter	Description	Default
FACE_MAX_CHANNELS_PER_MANAGER	Default maximum number of channels per channel manager.	50
MAX_CALLBACKS_PER_CONNECTION	Maximum number of callbacks per connection.	32
FACE_DEFAULT_BUFFER_SIZE	Buffer size for general-purpose operations (log messages, temporary buffers, etc.).	256
FACE_DEFAULT_MESSAGE_SIZE	Default message size for DDS message buffers.	1024
FACE_LARGE_BUFFER_SIZE	Large buffer size for configuration strings and bulk data.	8192
FACE_DEFAULT_DOMAIN_ID	Default DDS domain ID when none is configured.	0
FACE_DEFAULT_TIMEOUT_MS	Default timeout in milliseconds.	1000
FACE_RECEIVE_WAIT_SLICE_NS	Length in nanoseconds of one bounded wait slice. A blocking Receive_Message composes its caller's timeout from slices of this length rather than handing the whole timeout to the transport, so it can notice a closing connection between slices. The timeout a caller observes is unchanged; what this bounds is how long a teardown waits for a receive parked in the transport. Message latency is unaffected.	100000000
FACE_LOG_BUFFER_SIZE	Size of internal log message buffer.	4096
FACE_DEFAULT_HASH_TABLE_SIZE	Initial bucket count for hash tables.	128

The following hash table tuning parameters control the performance characteristics of internal hash tables used for connection, channel, and TPM lookups. Adjusting these values can trade memory usage for lookup speed:

Parameter	Description	Default
FACE_MIN_HASH_TABLE_SIZE	Minimum bucket count for hash tables. Tables will not shrink below this size.	64
HASH_TABLE_LOAD_FACTOR_PERCENT	Load factor threshold (as a percentage). When the ratio of entries to buckets exceeds this value, the table is resized. Lower values reduce collisions at the cost of more memory.	75
HASH_TABLE_GROWTH_FACTOR	Multiplier applied to the bucket count when the table is resized. A factor of 2 doubles the table on each resize.	2
HASH_MAX_ITERATIONS	Maximum probe iterations for a single hash operation. Prevents unbounded search times in degenerate cases.	100
DJB2_HASH_INIT	Initial seed value for the DJB2 hash algorithm.	5381
DJB2_HASH_MULT	Multiplier used in the DJB2 hash calculation (`hash = hash * mult + c`).	33
FNV_OFFSET_BASIS	Offset basis for the FNV-1a hash algorithm.	0x811c9dc5
FNV_PRIME	Prime multiplier for the FNV-1a hash algorithm.	0x01000193

Note: For SafetyBase (and stricter) profiles, dynamic memory allocation is disabled. In these profiles the resource limits set by these parameters determine the fixed, pre-allocated capacity of the system. Choosing values appropriate for your deployment is especially important.

3.4 Verifying Built Libraries

A successful build will create multiple libraries depending on which CMake flags were set. If you build C++ libraries, both C and C++ libraries will be present. All created libraries should be located in `<RTITSSHOME>/lib/<tss_arch>/<library_name>`:

- `librti_tssc` / `librti_tsscpp`: Type Abstraction library.
- `librti_tpm_micro_c` / `librti_tpm_micro_cpp`: Transport Protocol Module library based on *Connext Micro*.
- `librti_tpm_pro_c`: Transport Protocol Module library based on *Connext Professional*.

  Note: *Connext TSS* on *Connext Professional* does not support RTI Connext TSS C++ API.

- `librti_tss_serialization_c` / `librti_tss_serialization_cpp`: FACE-to-CDR serialization support library.

  Note:

  - Serialization support libraries are only used with *Connext Micro*.
  - The serialization libraries are built by default (`-DBUILD_SERIALIZATION=ON`), but they are **not** automatically linked by application build scripts (including the provided example `CMakeLists.txt`).
  - If your application uses FACE-to-CDR serialization, you must explicitly add `librti_tss_serialization_c` (C API) or `librti_tss_serialization_cpp` (C++ API) to the `target_link_libraries` call in your application's `CMakeLists.txt`.

  - If serialization is not needed, you can disable building the libraries by passing `-DBUILD_SERIALIZATION=OFF` to CMake when building *Connext TSS*.

- `librti_tss_strings_sequencesc`: Strings and sequences library.

Note: Static versions of the libraries have the letter z postpended to the filename. Debug versions of the libraries have the letter d postpended to the filename. For example, `librti_tsscz` is the release version of the static C API Type Abstraction library, and `librti_tsscdz` is the debug version of the static C API Type Abstraction library. Therefore, you can build both release and debug versions of the libraries in the same build directory.

4 Getting Started

After *installing* and *building* *Connext TSS* for your target architecture, you can run the provided example FACE application to understand how to use the *Connext TSS* libraries with a FACE application.

Note: This example provides a basic FACE logger that prints any log message sent to it, regardless of logging level. You can choose to modify the example or provide your own logger. For more information about logging, see *the API Reference*.

4.1 Building and Running the Example Application

Connext TSS ships with an example application that runs with *Connext Micro* or *Connext Professional*. The example, `hello_goodbye`, demonstrates how a TSS integrator can package additional functionality (plus the *Connext TSS* plugins) into a linkable library that can be used by a FACE Unit of Portability (UoP). It also shows how to combine multiple connections in the same application and how to set up a callback to process received data. The application includes only the minimum necessary code to build a FACE UoP using *Connext TSS*.

The following sections step you through how to build and run the example for Linux and Windows using C (*Connext Micro* or *Connext Professional*) or C++ (*Connext Micro*). To build the example for LynxOS or VxWorks, refer to the readme file for your language and OS at `<RTITSSHOME>\examples`. Detailed information about the example source code is included in the readme for each supported OS.

Note: *Connext TSS* supports user data types in IDL files only. To convert other FACE data model formats to IDL, contact RTI Support or your local account team for assistance.

4.1.1 Building the example

The `hello_goodbye` example contains a CMake script to generate the necessary plugins and build the application.

The example can be compiled against *Connext Micro* (GeneralPurpose and SafetyBase profiles) or *Connext Professional* (GeneralPurpose only). Out of the box, an application compiled with the GeneralPurpose profile will not communicate with an application compiled with the SafetyBase profile.

When invoking CMake, use the same variables as when *building the TSS libraries*.

- `RTI_CONNEXT_TYPE`
- `RTI_TSS_ENABLE_FACE_COMPLIANCE`
- `CMAKE_BUILD_TYPE`

- BUILD_SHARED_LIBS (optional)

1. To build the example, run the following commands:

```
cd ${RTITSSHOME}/examples/<C_Examples|C++_Examples>/<micro|pro>/<Linux|Windows>/
↪hello_goodbye
mkdir build
cd build
cmake -DRTI_CONNEXT_TYPE=<micro|pro> -DRTI_TSS_ENABLE_FACE_COMPLIANCE=
↪<GeneralPurpose|SafetyBase> -DCMAKE_BUILD_TYPE=<Release|Debug> ../
cmake --build .
```

Note: When building for Windows systems, Visual Studio will default to *Debug* even when *Release* is specified in the first CMake command above. To build *Release*, add `--config Release` to the second CMake command above.

Alternatively, use `cmake-gui` to configure additional CMake settings.

```
cd ${RTITSSHOME}/examples/<C_Examples|C++_Examples>/<micro|pro>/<Linux_or_
↪Windows>/hello_goodbye
mkdir build_gui
cd build_gui
cmake-gui ../
cmake --build .
```

2. The plugins are generated in the `gen` directory. The resulting executable is `../bin/${RTITSSARCH}/${RTI_CONNEXT_TYPE}/${CMAKE_BUILD_TYPE}/hello_goodbye_app`.

Each build is for a single combination of *Connext Micro*/*Connext Professional*, FACE profile, and CMake build type. Additional builds for different configuration combinations are best done out-of-source (i.e., from a directory separate from the source, like the `build` or `build-gui` directories created in step 1 above). However, note that the resulting applications are copied to the same output directory.

Note: By default, each run of CMake to build this example calls *rtiddsgen* to replace, or overwrite, all files in the `gen` directory.

To keep the generated source from being replaced, use the CMake command-line definition `-DRTI_REPLACE_GEN=false`.

4.1.2 Running the example

At this point, you've built the TSS library for *Connext Micro* or *Connext Professional* and linked it with the `hello_goodbye` example. Next, you'll run the example as a *Publisher* and as a *Subscriber*.

By default, the example must be run from the `hello_goodbye` directory where the default `TSS.xml` TSS configuration file is located. If you want to run the example from a different directory, set the XML file location in the corresponding configuration files in the `app` directory.

Running as a publisher

To run the example as a *Publisher*, launch the executable with the `-pub` command-line option:

```
./bin/${RTITSSARCH}/${RTI_CONNEXT_TYPE}/${CMAKE_BUILD_TYPE}/hello_goodbye_app -pub
```

There is one optional argument, `-num <NUM SAMPLES>`; this is the number of messages to send/receive before exiting.

The *Publisher* output will look similar to this:

```
--------------------------------------------------
 RTI Connext TSS HelloWorld Example (Connext Micro)
--------------------------------------------------
Sending unlimited samples ...
sending message 0
sending message 1
sending message 2
sending message 3
sending message 4
sending message 5
```

Running as a subscriber

To run the example as a *Subscriber*, use the `-sub` command-line option:

```
./bin/${RTITSSARCH}/${RTI_CONNEXT_TYPE}/${CMAKE_BUILD_TYPE}/hello_goodbye_app -sub
```

There is one optional argument, `-num <NUM SAMPLES>`: this is the number of messages to send/receive before exiting.

The output of a *Subscriber* doing polling reads looks similar to this:

```
--------------------------------------------------
 RTI Connext TSS HelloWorld Example (Connext Micro)
--------------------------------------------------
Receiving unlimited samples ...
received data: id[1] msg[Hello World 1]
received data: id[0] msg[Hello World 2]
received data: id[1] msg[Hello World 3]
received data: id[0] msg[Hello World 4]
received data: id[1] msg[Hello World 5]
```

Note that if there is no data available, the `Receive_Message` call will return a `FACE::NOT_AVAILABLE` error (ordinal value is 2), but the example will continue polling without shutting down.

Alternatively, to run a *Subscriber* in callback mode, use the `+sub` command-line option:

```
./bin/${RTITSSARCH}/${RTI_CONNEXT_TYPE}/${CMAKE_BUILD_TYPE}/hello_goodbye_app +sub
```

Its output will look similar to this:

```
-------------------------------------------------
RTI Connext TSS HelloWorld Example (Connext Micro)
-------------------------------------------------
Receiving unlimited samples ...
[FACE::DM::RTI::HelloWorld Callback_Handler called]
[FACE::DM::RTI::HelloWorld Callback_Handler called]
[FACE::DM::RTI::HelloWorld Callback_Handler called]
```

Note that the generated callback handler (in `gen/FACE/TSS/HelloWorld/TypedTS_Impl.c` or `gen/FACE/TSS/HelloWorld/TypedTS_Impl.cpp`) prints the same default message for each received sample. The generated callback handler(s) should be manually modified as necessary for your application.

Running as both publisher and subscriber

Note: This option is only available in *Connext Micro* examples.

To run both a *Publisher* and *Subscriber* within the same process, use the `-pubsub` command-line option:

```
./bin/${RTITSSARCH}/${RTI_CONNEXT_TYPE}/${CMAKE_BUILD_TYPE}/hello_goodbye_app -pubsub
```

This mode initializes the TSS stack once, creates separate publisher and subscriber connections (each with its own connection ID), and spawns two POSIX threads — one for publishing and one for subscribing — that run concurrently.

Use `-num <NUM SAMPLES>` to limit the number of messages sent and received:

```
./bin/${RTITSSARCH}/${RTI_CONNEXT_TYPE}/${CMAKE_BUILD_TYPE}/hello_goodbye_app -pubsub␣
↪-num 10
```

The output will show interleaved send and receive messages similar to this:

```
-------------------------------------------------
 RTI Connext TSS HelloWorld Example (Connext Micro)
-------------------------------------------------
sending message 0
sending message 1
received data: id[1] msg[Hello World 0] ...
sending message 2
received data: id[0] msg[Hello World 1] ...
```

If `-num` is not specified, both the publisher and subscriber will run indefinitely until the process is interrupted (e.g., with Ctrl+C).

4.2 Modifying the Example

Optionally, you can modify the example bundled with *Connext TSS* as noted below.

4.2.1 Changing QoS

To change the QoS for the underlying DDS *Entities*, configure the generated `_TSSQosSupport.cpp` or `_TSSQosSupport.c` source file as needed.

4.2.2 Adding connections

Each new connection creates an underlying DDS *DataWriter* and/or *DataReader*, so resource limit QoS settings may need to be increased for additional connections. See the RESOURCE_LIMITS QoS chapter in the Core Libraries User's Manual for more information.

4.3 Troubleshooting the Example

This section has tips for working around common issues.

4.3.1 Function returned an error

The example applications may return an error message in the following format:

```
<function> returned an error: <return_code>
```

For example, the following error message means `FACE::TS::Base::Receive_Message()` returned a `FACE::RETURN_CODE_TYPE` value of 2.

```
Receive_Message returned an error: 2
```

The return code can be found in a header file, and the return code of each function can be found in either source or the *API Reference* documentation.

For the above example, searching the headers in `include/FACE` reveals the `FACE::RETURN_CODE_TYPE` enum in `include/FACE/Common.hpp` or `include/FACE/Common.h`, where 2 maps to `NOT_AVAILABLE`:

```
typedef enum RETURN_CODE_TYPE
  {
    NO_ERROR ,
    NO_ACTION ,
    NOT_AVAILABLE ,
    INVALID_PARAM ,
    INVALID_CONFIG ,
    INVALID_MODE ,
    TIMED_OUT ,
    ADDR_IN_USE ,
    PERMISSION_DENIED ,
    MESSAGE_STALE ,
    IN_PROGRESS ,
    CONNECTION_CLOSED ,
    DATA_BUFFER_TOO_SMALL ,
    DATA_OVERFLOW ,
    RESOURCE_LIMIT_REACHED
  } RETURN_CODE_TYPE;
```

5 Porting Connext TSS

To build *Connext TSS* for a new platform or architecture, you must create or update files in the CMake resource directory (<RTITSSHOME>/resource/cmake).

The CMake resource directory contains the Toolchains/ and Modules/ directories. Toolchains/ contains CMake toolchain files. Modules/ contains RTI-specific CMake scripts, including additional platform- and toolchain-specific files.

5.1 New CMake Platforms

To port *Connext TSS* to a new CMake platform, a corresponding platform file may be required in the Toolchains/ directory (<RTITSSHOME>/resource/cmake/Toolchains/Platform). While a host platform likely will not need one, a cross-compiled platform likely will. Follow the CMake toolchain documentation to create a new platform file. Alternately, contact RTI Support and ask about the availability of new CMake platforms or toolchains.

To set RTI-specific compiler options, each platform must have a CMake platform file in the Modules directory (<RTITSSHOME>/resource/cmake/Modules/ProjectOptions/Platform/). These compiler options depend on the platform defined by the CMAKE_SYSTEM_NAME variable in a corresponding toolchain file (or guessed by CMake). RTI includes these platform and toolchain files in the <RTITSSHOME> top-level CMakeList file, based on the *RTI TSS architecture name*. These files may enable or disable *Connext TSS* features and products, depending on the CMake variables defined by you or the toolchain file.

5.1.1 Cross-compiling

A cross-compiled CMake platform must have another platform file in <RTITSSHOME>/resource/cmake/Modules/Platform/ to set the target platform. You must create a file named <CMAKE_SYSTEM_NAME>.cmake for this purpose; other configuration files are optional.

The following rules apply for the cross-compiled platform file:

- The file must start with the name of the platform to support, as defined by the CMAKE_SYSTEM_NAME variable.
- These files must **not** contain any RTI-specific options; those options need to be defined in the ProjectOptions/ folder (<RTITSSHOME>/resource/cmake/Modules/ProjectOptions/).
- Hard-coded paths are not allowed; use relative paths to CMake variables or environment variables.

5.2 New CMake Architectures

While the `Platform/` directories contain CMake files with shared settings between the toolchains of a platform family, the `Architecture/` directory (`<RTITSSHOME>/resource/cmake/Toolchains/Architecture/`) contains the toolchain file for a specific architecture. To port *Connext TSS* to a new CMake architecture, create a new toolchain file in this directory.

A basic toolchain file, for example, could include just the following:

```
# Toolchain file for Linux x64 with GCC 4.8.2

include("${CMAKE_CURRENT_LIST_DIR}/../Common.cmake")
platform_setup()

set(PLATFORM_C_COMPILER_EXPECTED_VERSION 4.8.2)
set(PLATFORM_CXX_COMPILER_EXPECTED_VERSION 4.8.2)

set(CMAKE_C_FLAGS "-m64 ${COMPILER_WARNING_FLAGS_C}")
set(CMAKE_CXX_FLAGS "${COMPILER_WARNING_FLAGS_CXX}")
set(CMAKE_EXE_LINKER_FLAGS "-static-libgcc ${CMAKE_EXE_LINKER_FLAGS}")

set(SYSLIBS_ARCH "-ldl -lnsl -lm -lpthread -lrt")
```

5.2.1 Variable definitions

General

The following variables can be used with any CMake architecture:

- `CMAKE_<LANG>_COMPILER`: Full path for the compiler.
- `CMAKE_AR`: Full path for the archiving tool for static libraries.
- `CMAKE_RANLIB`: Full path for the randomizing tool for static libraries.
- `CMAKE_LINKER`: Full path for the linker. Note that for most platforms, CMake uses the compiler for linking. This can be changed in the platform files, overriding the linking rules.
- `CMAKE_<LANG>_FLAGS_INIT`: Flags for the compiler.
- `CMAKE_SHARED_LINKER_FLAGS_INIT`: Flags for the linker.
- `COMPILER_WARNING_FLAGS_C`: Flags to enable warnings in the C compiler.
- `COMPILER_WARNING_FLAGS_CXX`: Flags to enable warnings in the C++ compiler.
- `WARNING_AS_ERROR_FLAG`: Flags to enable treat warnings as errors.

Typically, it's enough to specify the compilers in your toolchain file. CMake internally searches for the tools *ar* and *ranlib*, as well as the *linker*, from the directory containing the compiler.

Cross-compiling

The following CMake variables can be used when cross-compiling:

- `CMAKE_SYSTEM_NAME`: Target platform name, usually output from `uname -s`. Used by CMake to load the platform configuration files under *Platform/$<CMAKE_SYSTEM_NAME>*. Setting this variable means cross-compiling and CMake sets the `CMAKE_CROSSCOMPILING` variable.
- `CMAKE_SYSTEM_VERSION`: Target platform version, usually output from `uname -r`. Not used by CMake.
- `CMAKE_SYSTEM_PROCESSOR`: Target CPU name, usually output from `uname -p`. Used to load options for the compiler.
- `CMAKE_SYSROOT`: Path to pass to the compiler in the `--sysroot` flag.

6 Capabilities and Interfaces

6.1 Programming Language Support

Connext TSS supports the FACE TSS APIs in C++ and C.

6.2 Operating System Environment

Connext TSS has been implemented using the POSIX API calls prescribed for the applicable conformance profile.

6.3 TSS Capabilities

Connext TSS supports the following capabilities:

- Transport Service (TS)
- TSS Distribution
- TSS Configuration
- Type Abstraction
- Transport Protocol Module

Connext TSS generates support code with the included *Code Generator* (*rtiddsgen*) tool to enable these capabilities.

6.4 FACE Profiles

Connext TSS officially supports two FACE Compliance profiles, GeneralPurpose and SafetyBase. The `RTI_TSS_ENABLE_FACE_COMPLIANCE` CMake definition can be set to any FACE TSS profile. However, check the *Supported Platforms* section for information on which profiles are supported per platform.

6.4.1 GeneralPurpose profile

Connext TSS supports the GeneralPurpose Profile with *Connext Micro* and *Connext Professional*. While `RTI_TSS_ENABLE_FACE_COMPLIANCE` can be set to either GeneralPurpose or None, both are mapped to the GeneralPurpose Profile.

Run the following CMake command to build *Connext TSS* for the GeneralPurpose Profile with *Connext Micro*:

```
cmake -DRTI_CONNEXT_TYPE=micro -DRTI_TSS_ENABLE_FACE_
↪COMPLIANCE=GeneralPurpose -DCMAKE_BUILD_TYPE=Release ../
```

To build *Connext TSS* with the GeneralPurpose profile for *Connext Professional* instead, run the following command:

```
cmake -DRTI_CONNEXT_TYPE=pro -DRTI_TSS_ENABLE_FACE_
↪COMPLIANCE=GeneralPurpose -DCMAKE_BUILD_TYPE=Release ../
```

Note: Refer to the *Building Connext TSS* chapter for a step-by-step guide to building *Connext TSS* libraries.

6.4.2 SafetyBase profile

Connext TSS also supports the SafetyBase Profile with *Connext Micro*. While `RTI_TSS_ENABLE_FACE_COMPLIANCE` can be set to either SafetyExtended, SafetyBase, or Security, all three are mapped to the SafetyBase Profile because *Connext TSS* does not specifically support SafetyExtended or Security.

Run the following CMake command to build *Connext TSS* (Release) for the SafetyBase Profile:

```
cmake -DRTI_CONNEXT_TYPE=micro -DRTI_TSS_ENABLE_FACE_COMPLIANCE=SafetyBase -
↪DCMAKE_BUILD_TYPE=Release ../
```

Note: *Connext Micro* is the only option for `-DRTI_CONNEXT_TYPE` when building for the SafetyBase profile.

6.4.3 Limitations

Connext TSS built for the SafetyBase profile is different from TSS built for GeneralPurpose. *Connext TSS* SafetyBase is limited to only using DPSE discovery. For details on DPSE discovery, see the Discovery section of the *Connext Micro* documentation, located in the `doc` folder of your *Connext Micro* installation directory (`<RTIMEHOME>`).

Connext TSS SafetyBase also does not support multicast, octet, or internal logging. Most importantly, *Connext TSS* SafetyBase does not support deallocation or reallocation of allocated memory. This limitation extends to all RTI entities and resources.

6.5 Type Support

DDS is a strongly typed API that requires explicit definition of the data types to be used. The definition is provided in a platform- and language-neutral description such as IDL.

Note: *Connext TSS* supports user data types in IDL files only. FACE requires that the user data types are in the FACE::DM namespace. In IDL, the module keyword is used to create namespaces for the declaration of types defined within the file. For example:

```
module FACE {
    module DM {
        struct <user data type> {};
    };
};
```

Warning: Declaring structs in IDL files outside of the FACE::DM module is not permitted by the FACE technical standard. However, *rtiddsgen* will not raise an error in this case; it is up to you to ensure that no structs are declared elsewhere.

Once this definition is provided, the *rtiddsgen* tool is used to generate type support code for the given data type(s). The generated code (for a specific language) contains the following definitions:

- the type itself
- the *DataWriter* and *DataReader* code
- serialization and deserialization code
- other utilities needed to use the data type in DDS systems

In a DDS application, the generated type support is "registered" with a *DomainParticipant* via an explicit call:

```
HelloWorldTypeSupport_register_type(participant, type_name);
```

Essentially, this function installs a type support plugin that registers the various type-specific utilities so that the DDS core can use the type.

The FACE TSS API is also strongly typed. The modified version of the *rtiddsgen* tool provided with *Connext TSS* generates both the type support and the type plugin required by *Connext TSS*.

The generated type plugin files (for example, HelloGoodbye, HelloGoodbyePlugin, and HelloGoodbyeSupport) should not need any modification. The `register_type` function for a specific type is configured from the *Configuration Interface* capability described below.

6.6 Type Abstraction Base and Typed interfaces

While the FACE 3.x TSS Type-Specific interface is defined and implemented by *Connext TSS* independently of any user data type, the TSS Type-Specific Typed interface must be defined for each type. Consequently, the *rtiddsgen* tool is used to generate the following headers and source files:

- `<MODULES/<DATATYPE_TYPE>/TypedTS.hpp` defines the Type-Specific Typed interface for the user data type.
- `<MODULES/<DATATYPE_TYPE>/TypedTS_Impl.hpp` defines a class implementating the interface defined in `<MODULES/<DATATYPE_TYPE>/TypedTS.hpp`.
- `<MODULES/<DATATYPE_TYPE>/TypedTS_Impl.cpp` is the implementation of `<MODULES/<DATATYPE_TYPE>/TypedTS_Impl.hpp`.

6.7 Configuring FACE 3.x for Connext Micro

Connext TSS utilizes the FACE 3.x Configuration Interface. Configure *Connext TSS* by customizing configuration data within a Configuration Interface instance and then setting its reference with the Type Abstraction or Transport Protocol Module. The Type Abstraction configuration data is XML-based and uses `name:value` string pairs to handle its configuration elements. The Transport Protocol Module configuration data is used by the lower level DDS product to configure its DDS settings. Transport Protocol Module utilizes configuration names set in the DDS data to handle its configuration.

6.7.1 Type Abstraction configuration implementation

Connext TSS provides example source code for a Configuration Interface. You need to replicate the XML configuration layout in your implementation of the Configuration Interface. *Connext TSS* provides the `tss_configuration_schema.xsd` file to validate the configuration XML, as well as an example `TSS.xml` configuration file.

Warning: The example configuration is provided as a baseline configuration. Modify the example configuration to adjust for your connections and Transport Protocol Modules.

If you already have a configuration UoC implemented, the following section describes the expected string format for the byte_array returned by `Read_Configuration()`. For example, `guid:5;` could be used as a configuration string for registering a Transport Protocol Module with Type Abstraction. A configuration string for a connection could be, for example, `tpm:5;tpm:5;tpm:7;`.

For each Transport Protocol Module to be registered with the Type Abstraction, the Type Abstraction expects to receive a configuration string in the returned buffer in the format `guid:<TPM GUID>` when `Read_Configuration()` is called to load a list of Transport Protocol Modules to be injected. For example, `guid:5;`. For each connection to be created with the Type Abstraction, the Type Abstraction expects a configuration string in the returned buffer in the format `tpm:<First TPM GUID>;tpm:<Second TPM GUID>;...tpm:<Nth TPM GUID>;`, where each Transport Protocol Module that is going to participate on the connection needs to be identified by its GUID. For example, `tpm:5;tpm:5;tpm:7;`.

Type Abstraction XML configuration elements

Connext TSS includes a set of XML elements that are used to configure the Type Abstraction. The xsd for those elements can be found in `<RTITSSHOME>/resource/configuration/schema/tss_configuration_schema.xsd`. The elements are:

- configs

  This is the root element of the configuration file. It contains all the configuration elements for a system.

  - config

    This element contains all the configuration data for a specific configuration. It has a required name attribute that is used to identify the configuration. This attribute must match what is passed into the `Configuration_Open()` call in the code. It contains the following elements:

    * connections

      This element contains a list of the connections for a specific configuration.

      · connection

      This element contains the connection configuration details for a specific connection in the Type Abstraction. It has a required name attribute that is used to identify the connection. The name attribute must match what is passed into the `Create_Connection` call in the code. It contains the following elements:

- tpms

  This element contains a list of the Transport Protocol Modules for a specific connection.

  - tpm

    This element contains Transport Protocol Module information for the specific connection. It has a required guid attribute that is used to identify the Transport Protocol Module. The guid attribute must match a Transport Protocol Module guid in the list of Transport Protocol Modules under the tpms element under the config element

* tpms

  This element contains the Transport Protocol Modules for a specific configuration.

  - tpm

    This element contains the configuration data for a specific Transport Protocol Module that's needed by the Type Abstraction. It has required guid and name attributes that are used to identify the Transport Protocol Module. Each name and guid must be unique per configuration.

* types

  This element contains a list of the types for a specific configuration.

  Note: The types element is not used in the current implementation of *Connext TSS*.

  - type

    This element contains all the type elements for a specific type. It has a required name attribute that is used to identify the type.

6.7.2 Transport Protocol Module configuration implementation

Transport Protocol Module provides example source code for a Configuration Interface. You need to copy the HelloWorld example configuration for your language and implement the `FACE::Configuration` interface (if using C++) or the `FACE_Configuration` interface (if using C) and the interface's Configuration Services API. Examples and descriptions of what is required are in the following sections.

Warning: The example configuration is provided as a baseline configuration. You will need to modify the example Configuration Interface implementations and QoS functions.

6.7.3 OSAPI configuration for Connext Micro

When building Transport Protocol Module with *Connext Micro*, the OSAPI is used to allocate memory during the initialization phase. OSAPI must be set up before calling any of the functions it provides. To do so, the example code calls the `_set_osapi_property` function, followed by the `DDS_DomainParticipantFactory_get_instance` function.

Example OSAPI configuration in C++:

```
void Initialize(const FACE::Configuration::INITIALIZATION_TYPE &initialization_
↪information,
                FACE::RETURN_CODE_TYPE::Value &return_code)
{
    UNUSED_ARG(initialization_information);

    if (_initialized)
```

(continues on next page)

(continued from previous page)

```
    {
        return_code = FACE::RETURN_CODE_TYPE::NO_ERROR;
        return;
    }

    #ifdef RTI_CONNEXT_MICRO

    /* OSAPI needs to be set up before it can be used. */
    if (!_set_osapi_property())
    {
        return_code = FACE::RETURN_CODE_TYPE::INVALID_CONFIG;
        return;
    }

    DDS_DomainParticipantFactory *factory = NULL;
    factory = DDS_DomainParticipantFactory_get_instance();
    if (factory == NULL)
    {
        return_code = FACE::RETURN_CODE_TYPE::INVALID_CONFIG;
        return;
    }
    /* If and only if the set up above are successfully,
    *  OSAPI is ready to be used after this point.
    */

    #endif

    if (!_populate_system_config() ||
        !_populate_domain_config() ||
        !_populate_type_support_config() ||
        !_populate_connection_config())
    {
        return_code = FACE::RETURN_CODE_TYPE::INVALID_CONFIG;
        return;
    }

    _initialized = DDS_BOOLEAN_TRUE;
    return_code = FACE::RETURN_CODE_TYPE::NO_ERROR;
}
```

Example OSAPI configuration in C:

```
FACE_interface_return FACE_Configuration_Initialize(
                struct FACE_Configuration* this_obj,
                const FACE_Configuration_INITIALIZATION_TYPE *initialization_
↪information,
                FACE_RETURN_CODE_TYPE *return_code)
 {
      if(this_obj == NULL)
      {
          return FACE_INTERFACE_NULL_PARAM;
      }

      UNUSED_ARG(initialization_information);

      if (_initialized)
```

(continues on next page)

(continued from previous page)

```
    {
        *return_code = FACE_RETURN_CODE_TYPE_NO_ERROR;
        return FACE_INTERFACE_NO_ERROR;
    }

    #ifdef RTI_CONNEXT_MICRO

        /* OSAPI needs to be set up before it can be used. OSAPI
        should only be called once by the program otherwise undefined
        behavior may occur.*/
        if (!_set_osapi_property())
        {
            *return_code = FACE_RETURN_CODE_TYPE_INVALID_CONFIG;
            return FACE_INTERFACE_NO_ERROR;
        }

        DDS_DomainParticipantFactory *factory = NULL;
        factory = DDS_DomainParticipantFactory_get_instance();
        if (factory == NULL)
        {
            *return_code = FACE_RETURN_CODE_TYPE_INVALID_CONFIG;
            return FACE_INTERFACE_NO_ERROR;
        }
        /* If and only if the set up above is successful,
        *  OSAPI can be used after this point.
        */
    #endif
    if (!_populate_system_config() ||
        !_populate_domain_config() ||
        !_populate_type_support_config() ||
        !_populate_connection_config())
    {
        *return_code = FACE_RETURN_CODE_TYPE_INVALID_CONFIG;
        return FACE_INTERFACE_NO_ERROR;
    }

    _initialized = DDS_BOOLEAN_TRUE;
    *return_code = FACE_RETURN_CODE_TYPE_NO_ERROR;
        return FACE_INTERFACE_NO_ERROR;
}
```

To modify the OSAPI properties, you can override the `_set_osapi_property` function instead of creating a plugin function, then pass it to the now-deprecated `RTI_TSS_System_Configuration.configure_osapi_system_property_fn` *Connext Micro* function (this function does nothing by default).

Use the following steps to override the `_set_osapi_property` function:

1. Create a variable of struct `OSAPI_SystemProperty` and initialize it with `OSAPI_SystemProperty_INITIALIZER`.
2. Get the current OSAPI properties by passing the variable to the `OSAPI_System_get_property` function.
3. Modify the `OSAPI_SystemProperty` variable. Notable properties include:
    - `OSAPI_SystemProperty.timer_property.thread.port_property.parent.period`
    - `OSAPI_SystemProperty.timer_property.thread.port_property.parent.time_capacity`
    - `OSAPI_SystemProperty.timer_property.thread.port_property.parent.name`
    - `OSAPI_SystemProperty.timer_property.thread.port_property.parent.deadline`

- `OSAPI_SystemProperty.timer_property.thread.stack_size`
- `OSAPI_SystemProperty.timer_property.thread.priority`

4. Pass the modified `OSAPI_SystemProperty` variable to `OSAPI_System_set_property`.

6.7.4 Configuration data types

The configuration data types of the `RTI::Configuration` class are defined in the file `code/C/include/RTI/TSS/Configuration.h`. They are organized in the following categories:

- System Configuration (`RTI_TSS_System_Configuration_T`)
- *DomainParticipant* Configuration (`RTI_TSS_DDS_DomainParticipant_Configuration_T`)
- Type Configuration (`RTI_TSS_TypeSupport_Configuration_T`)
- Connection Configuration (`RTI_TSS_Channel_Configuration_t`)

System configuration

The system configuration sets system properties for Transport Protocol Module and the underlying DDS factories. It is applied upon the very first call, idempotently, to `FACE::TSS::TPM_TPMTS::Initialize()` (if using C++) or `FACE_TSS_TPM_TPMTS_Initialize()` (if using C). A system configuration is represented by `RTI_TSS_System_Configuration_T`.

Example system configuration in C/C++:

```
/*RTI_TSS_System_Configuration */
typedef struct RTI_TSS_System_Configuration
{
    /*Name of the System Configuration */
    const char * config_name;

#if not in a Safety Base profile
    /*Logging verbosity */
    LogVerbosity verbosity;
#endif

    /*the configure_domain_participant_factory_qos_fn that configures the domain␣
↪participant factory*/
    RTI_TSS_DomainParticipantFactoryQosCallback
            configure_domain_participant_factory_qos_fn;

    /*additional configuration data */
    void *user_data;

#if using RTI Connext Micro
    /*max number of connections that are allowed in the system */
    RTI_UINT32 max_channels;

    /*max number of topics that are allowed in the system */
    RTI_UINT32 max_topics;

    /*pointer to an array of the Micro components used for configuring the low level␣
↪Micro Settings*/
    RTI_TSS_Micro_Component_T *components;

    /*number of components */
```

(continues on next page)

(continued from previous page)

```
    RTI_UINT32 components_length;

#endif
#if using RTI Connext Pro
    /*pointer to the xml qos file */
    const char *xml_qos_file;

    /*pointer to the qos_library to be used by the system */
    const char* qos_library;

    /*pointer to the qos_profile to be used by the system*/
    const char* qos_profile;
#endif
} RTI_TSS_System_Configuration_T;
```

The example RTI configuration class defines a function, `_populate_system_config()`, that sets a static array of system configurations that are read by a Base instance when it is initialized with the Configuration Interface.

Example system configuration snippet in C++:

```
/* Example system configuration snippet from a Foo_TSSConfigInterface.hpp */

static const FACE::UnsignedLong _system_configurations_length = 2;
RTI_TSS_System_Configuration_T _system_configurations[_system_configurations_length];

FACE::Boolean _populate_system_config()
{
    /* Customize each element of _system_configurations[] with a system configuration
     * that may be used to configure your Connext TSS application.
     */
    if (!RTI_TSS_System_Configuration_initialize(&_system_configurations[0]))
    {
        return false;
    }
    _system_configurations[0].config_name = "HelloWorld";
    ...

    if (!RTI_TSS_System_Configuration_initialize(&_system_configurations[1]))
    {
        return false;
    }
    _system_configurations[1].config_name = "GoodbyeWorld";
    ...
}
```

Example system configuration snippet in C:

```
/* Example system configuration snippet from a Foo_TSSConfigInterface.hpp */

static const FACE_unsigned_long _system_configurations_length = 2;
RTI_TSS_System_Configuration_T _system_configurations[_system_configurations_length];

FACE_boolean _populate_system_config()
{
    /* Customize each element of _system_configurations[] with a system configuration
     * that may be used to configure your Connext TSS application.
     */
```

(continues on next page)

(continued from previous page)

```
    if (!RTI_TSS_System_Configuration_initialize(&_system_configurations[0]))
    {
        return false;
    }
    _system_configurations[0].config_name = "HelloWorld";
    ...

    if (!RTI_TSS_System_Configuration_initialize(&_system_configurations[1]))
    {
        return false;
    }
    _system_configurations[1].config_name = "GoodbyeWorld";
    ...
}
```

Connext Micro RT components

For *Connext Micro* (or *Connext Cert*), run-time (RT) components must be configured and registered during system initialization. These components for *Connext Micro* include transports (UDP, ARINC 653) and discovery (DPDE, DPSE). To support their configuration, the system configuration type contains an array of configurations for *Connext Micro* RT components, all of which are registered with *Connext Micro* upon initialization.

Because the property of each component is complicated to set, the example configuration provides functions for you to customize that return the component property.

The snippet below shows how components are configured for a system configuration. Note how the function `_new_udp_property()` is defined and configured to return a `UDP_InterfaceFactoryProperty` for the UDP component. This pattern is applied for other components as well.

Example in C++:

```
/* Example Micro components snippet from a Foo_TSSConfigInterface.hpp */

FACE::Boolean _populate_system_config()
{
    ...
    /* Customize the Micro run-time components for _system_configurations[0] */
    _system_configurations[0].components_length = 4;
    _system_configurations[0].components =
        new RTI_TSS_Micro_Component_t[_system_configurations[0].components_length];

    // Writer History Component
    if (!RTI_TSS_Micro_Component_initialize(&_system_configurations[0].components[0]))
    {
        return false;
    }
    _system_configurations[0].components[0].name = DDSHST_WRITER_DEFAULT_HISTORY_NAME;
    _system_configurations[0].components[0].component = WHSM_HistoryFactory_get_
↪interface();
    _system_configurations[0].components[0].listener = NULL;
    _system_configurations[0].components[0].property = NULL;

    // Reader History Component
    if (!RTI_TSS_Micro_Component_initialize(&_system_configurations[0].components[1]))
    {
        return false;
```

(continues on next page)

(continued from previous page)

```
    }
    _system_configurations[0].components[1].name = DDSHST_READER_DEFAULT_HISTORY_NAME;
    _system_configurations[0].components[1].component = RHSM_HistoryFactory_get_
↪interface();
    _system_configurations[0].components[1].listener = NULL;
    _system_configurations[0].components[1].property = NULL;

    // UDP Component
    if (!RTI_TSS_Micro_Component_initialize(&_system_configurations[0].components[2]))
    {
        return false;
    }
    _system_configurations[0].components[2].name = NETIO_DEFAULT_UDP_NAME;
    _system_configurations[0].components[2].component = UDP_InterfaceFactory_get_
↪interface();
    _system_configurations[0].components[2].listener = NULL;
    _system_configurations[0].components[2].property =
            (struct RT_ComponentFactoryProperty*) _new_udp_property();

      // DPSE Component
      struct DPSE_DiscoveryPluginProperty *dpse_properties =
              (struct DPSE_DiscoveryPluginProperty*) malloc(
                      sizeof(struct DPSE_DiscoveryPluginProperty));

      if (DPSE_DiscoveryPluginProperty_initialize(dpse_properties) != DDS_RETCODE_OK)
      {
          printf("Failed to Initialize DPSE properties");
          return false;
      }

      if (!RTI_TSS_Micro_Component_initialize(&_system_configurations[0].
↪components[3]))
      {
          return false;
      }
      _system_configurations[0].components[3].name = "dpse";
      _system_configurations[0].components[3].component = DPSE_DiscoveryFactory_get_
↪interface();
      _system_configurations[0].components[3].listener = NULL;
      _system_configurations[0].components[3].property =
              (struct RT_ComponentFactoryProperty*) dpse_properties;

      ...
}

/* Customize the UDP property for the Micro UDP transport component */
struct UDP_InterfaceFactoryProperty *_new_udp_property()
{
    struct UDP_InterfaceFactoryProperty *udp_property =
        (struct UDP_InterfaceFactoryProperty *) malloc(
                    sizeof(struct UDP_InterfaceFactoryProperty));
    memset(udp_property, 0, sizeof(struct UDP_InterfaceFactoryProperty));
    *udp_property = UDP_INTERFACE_FACTORY_PROPERTY_DEFAULT;

    udp_property->disable_auto_interface_config = RTI_TRUE;
```

(continues on next page)

(continued from previous page)

```
    REDA_StringSeq_set_maximum(&udp_property->allow_interface,1);
    REDA_StringSeq_set_length(&udp_property->allow_interface,1);

    *DDS_StringSeq_get_reference(&udp_property->allow_interface,0) =
         DDS_String_dup("loopback");

    if (!UDP_InterfaceTable_add_entry(&udp_property->if_table,
                                      0x7f000001,0xff000000,"loopback",
                                      UDP_INTERFACE_INTERFACE_UP_FLAG))
    {
        printf("Failed UDP table add entry!\n");
        return NULL;
    }

    ...
    return udp_property;
}
```

Example in C:

```
/* Example Micro components snippet from a Foo_TSSConfigInterface.hpp */

FACE_boolean _populate_system_config()
{
    ...
    /* Customize the Micro run-time components for _system_configurations[0] */
    _system_configurations[0].components_length = 4;
    _system_configurations[0].components =
        new RTI_TSS_Micro_Component_t[_system_configurations[0].components_length];

    // Writer History Component
    if (!RTI_TSS_Micro_Component_initialize(&_system_configurations[0].components[0]))
    {
        return false;
    }
    _system_configurations[0].components[0].name = DDSHST_WRITER_DEFAULT_HISTORY_NAME;
    _system_configurations[0].components[0].component = WHSM_HistoryFactory_get_
↪interface();
    _system_configurations[0].components[0].listener = NULL;
    _system_configurations[0].components[0].property = NULL;

    // Reader History Component
    if (!RTI_TSS_Micro_Component_initialize(&_system_configurations[0].components[1]))
    {
        return false;
    }
    _system_configurations[0].components[1].name = DDSHST_READER_DEFAULT_HISTORY_NAME;
    _system_configurations[0].components[1].component = RHSM_HistoryFactory_get_
↪interface();
    _system_configurations[0].components[1].listener = NULL;
    _system_configurations[0].components[1].property = NULL;

    // UDP Component
    if (!RTI_TSS_Micro_Component_initialize(&_system_configurations[0].components[2]))
    {
        return false;
```

(continues on next page)

(continued from previous page)

```
    }
    _system_configurations[0].components[2].name = NETIO_DEFAULT_UDP_NAME;
    _system_configurations[0].components[2].component = UDP_InterfaceFactory_get_
↪interface();
    _system_configurations[0].components[2].listener = NULL;
    _system_configurations[0].components[2].property =
            (struct RT_ComponentFactoryProperty*) _new_udp_property();

      // DPSE Component
      struct DPSE_DiscoveryPluginProperty *dpse_properties =
              (struct DPSE_DiscoveryPluginProperty*) malloc(
                      sizeof(struct DPSE_DiscoveryPluginProperty));

      if (DPSE_DiscoveryPluginProperty_initialize(dpse_properties) != DDS_RETCODE_OK)
      {
          printf("Failed to Initialize DPSE properties");
          return false;
      }

      if (!RTI_TSS_Micro_Component_initialize(&_system_configurations[0].
↪components[3]))
      {
          return false;
      }
      _system_configurations[0].components[3].name = "dpse";
      _system_configurations[0].components[3].component = DPSE_DiscoveryFactory_get_
↪interface();
      _system_configurations[0].components[3].listener = NULL;
      _system_configurations[0].components[3].property =
              (struct RT_ComponentFactoryProperty*) dpse_properties;

      ...
}

/* Customize the UDP property for the Micro UDP transport component */
struct UDP_InterfaceFactoryProperty *_new_udp_property()
{
    struct UDP_InterfaceFactoryProperty *udp_property =
        (struct UDP_InterfaceFactoryProperty *) malloc(
                        sizeof(struct UDP_InterfaceFactoryProperty));
    memset(udp_property, 0, sizeof(struct UDP_InterfaceFactoryProperty));
    *udp_property = UDP_INTERFACE_FACTORY_PROPERTY_DEFAULT;

    udp_property->disable_auto_interface_config = RTI_TRUE;

    REDA_StringSeq_set_maximum(&udp_property->allow_interface,1);
    REDA_StringSeq_set_length(&udp_property->allow_interface,1);

    *DDS_StringSeq_get_reference(&udp_property->allow_interface,0) =
         DDS_String_dup("loopback");

    if (!UDP_InterfaceTable_add_entry(&udp_property->if_table,
                                      0x7f000001,0xff000000,"loopback",
                                      UDP_INTERFACE_INTERFACE_UP_FLAG))
    {
        printf("Failed UDP table add entry!\n");
```

(continues on next page)

(continued from previous page)

```
        return NULL;
    }

    ...
    return udp_property;
}
```

Connext Micro ARINC 653 transport

Configuration of the ARINC 653 transport of *Connext Micro* requires registration of the RT components for the Port Manager and the ARINC Interface(s).

Connext Micro requires one Port Manager instance for each ARINC 653 partition in order to create APEX ports and manage their sending and receiving of messages.

Connext Micro also requires one ARINC 653 interface instance for each *DomainParticipant*.

The snippet below configures a Port Manager with one APEX send port and one APEX receive port and an ARINC Interface that uses the aforementioned Port Manager.

Example ARINC 653 snippet in C++:

```
/* Example ARINC 653 snippet from a Foo_TSSConfigInterface.hpp */

FACE::Boolean _populate_system_config()
{
    ...
    // ARINC 653 Port Manager
    if (!RTI_TSS_Micro_Component_initialize(&_system_configurations[0].components[5]))
    {
        return false;
    }
    _system_configurations[0].components[5].name = NETIO_DEFAULT_PORTMANAGER_NAME;
    _system_configurations[0].components[5].component = ARINC_PortManagerFactory_get_
↪interface();
    _system_configurations[0].components[5].listener = NULL;
    _system_configurations[0].components[5].property =
          (struct RT_ComponentFactoryProperty*) _new_prtmgr_property();

    // ARINC 653 Transport
    if (!RTI_TSS_Micro_Component_initialize(&_system_configurations[0].components[6]))
    {
      return false;
    }
    _system_configurations[0].components[6].name = NETIO_DEFAULT_ARINC_NAME;
    _system_configurations[0].components[6].component = ARINC_InterfaceFactory_get_
↪interface();
    _system_configurations[0].components[6].listener = NULL;
    _system_configurations[0].components[6].property =
          (struct RT_ComponentFactoryProperty*) _new_arinc_property();

     ...
}

struct ARINC_PortManagerFactoryProperty *_new_prtmgr_property()
{
     struct ARINC_RouteProperty *route_property = NULL;
```

(continues on next page)

(continued from previous page)

```
    struct ARINC_ReceivePortProperty *receive_port_prop = NULL;
    struct ARINC_PortManagerFactoryProperty *port_mgr_prop =
   (struct ARINC_PortManagerFactoryProperty *) malloc(
       sizeof(struct ARINC_PortManagerFactoryProperty));
   memset(port_mgr_prop, 0, sizeof(struct ARINC_PortManagerFactoryProperty));
   *port_mgr_prop = ARINC_PORTMANAGER_FACTORY_PROPERTY_DEFAULT;

   /* Configure synchronous or asynchronous processing.
    * Synchronous (TRUE) will use this Port Manager's receive thread to pass
    * a received message all the way up to the receiving application.
    * Asynchronous (FALSE) will enqueue the message with the ARINC Interface, which
    * then uses its receive thread to pass each message up to the receiving␣
↪application.
    */
    port_mgr_prop->sync_processing = RTI_TRUE;

   /* Configure APEX process attributes of the Port Manager's receive thread */
   port_mgr_prop->thread_property.port_property.parent.deadline = SOFT;
   port_mgr_prop->thread_property.port_property.parent.period = 100000000;
   port_mgr_prop->thread_property.port_property.parent.time_capacity = 100000000;
   port_mgr_prop->thread_property.stack_size = 32000;
   port_mgr_prop->thread_property.priority = 99;

   #define RTI_PORTMGR_RECV_THREAD_NAME "rti.ARINC.pm.rcv_thrd"
   OSAPI_Memory_zero(&port_mgr_prop->thread_property.port_property.parent.name, MAX_
↪NAME_LENGTH);
   OSAPI_Memory_copy(&port_mgr_prop->thread_property.port_property.parent.name,
       RTI_PORTMGR_RECV_THREAD_NAME,
       OSAPI_String_length(RTI_PORTMGR_RECV_THREAD_NAME));


   /* Configure send routes */
   ARINC_RoutePropertySeq_set_maximum(&port_mgr_prop->send_routes, 1);
   ARINC_RoutePropertySeq_set_length(&port_mgr_prop->send_routes, 1);
   route_property = ARINC_RoutePropertySeq_get_reference(
       &port_mgr_prop->send_routes, 0);

   /* Configure a route from an APEX queuing port sending via a channel
    * with a unique channel identifier.
    */
   #define RTI_APP_SEND_PORT_NAME "Part1_send"
   route_property->channel_identifier = 1;
   route_property->send_port_property.max_message_size = 1024;
   route_property->send_port_property.max_queue_size = 32;
   OSAPI_Memory_copy(route_property->send_port_property.port_name,
       RTI_APP_SEND_PORT_NAME,
       OSAPI_String_length(RTI_APP_SEND_PORT_NAME));

   /* Configure APEX queuing ports to receive */
   ARINC_ReceivePortPropertySeq_set_maximum(&port_mgr_prop->receive_ports,1);
   ARINC_ReceivePortPropertySeq_set_length(&port_mgr_prop->receive_ports,1);
   receive_port_prop = ARINC_ReceivePortPropertySeq_get_reference(
       &port_mgr_prop->receive_ports,0);

   /* Configure an APEX queuing port receiving via a channel with
    * a unique channel identifier. */
```

(continues on next page)

(continued from previous page)

```
    #define RTI_APP_RECV_PORT_NAME "Part1_recv"
    receive_port_prop->channel_identifier = 2;
    receive_port_prop->receive_port_property.max_message_size = 1024;
    receive_port_prop->receive_port_property.max_queue_size = 32;
    receive_port_prop->max_receive_queue_size = 32;
    OSAPI_Memory_copy(receive_port_prop->receive_port_property.port_name,
        RTI_APP_RECV_PORT_NAME,
        OSAPI_String_length(RTI_APP_RECV_PORT_NAME));

    return port_mgr_prop;
}

struct ARINC_InterfaceFactoryProperty *_new_arinc_property()
{
    struct ARINC_InterfaceFactoryProperty *arinc_property =
        (struct ARINC_InterfaceFactoryProperty *)
            malloc(sizeof(struct ARINC_InterfaceFactoryProperty));
    *arinc_property = ARINC_INTERFACE_FACTORY_PROPERTY_DEFAULT;

    /* Configure APEX process attributes of the ARINC Interface's receive thread.
     * Used only when this interface's Port Manager's sync_processing property is␣
↪FALSE.
     */
    arinc_property->recv_thread.port_property.parent.deadline = SOFT;
    arinc_property->recv_thread.port_property.parent.period = 100000000;
    arinc_property->recv_thread.port_property.parent.time_capacity = 100000000;
    arinc_property->recv_thread.stack_size = 16000;
    arinc_property->recv_thread.priority = 97;

    #define RTI_RECV_THREAD_NAME "rti.ARINC.p2.rcv_thrd"
    OSAPI_Memory_zero(&arinc_property->recv_thread.port_property.parent.name, MAX_
↪NAME_LENGTH);
    OSAPI_Memory_copy(&arinc_property->recv_thread.port_property.parent.name,
                      RTI_RECV_THREAD_NAME,
    OSAPI_String_length(RTI_RECV_THREAD_NAME));

    return arinc_property;
}
```

Example ARINC 653 snippet in C:

```
/* Example ARINC 653 snippet from a Foo_TSSConfigInterface.hpp */

FACE_boolean _populate_system_config()
{
    ...
    // ARINC 653 Port Manager
    if (!RTI_TSS_Micro_Component_initialize(&_system_configurations[0].components[5]))
    {
        return false;
    }
    _system_configurations[0].components[5].name = NETIO_DEFAULT_PORTMANAGER_NAME;
    _system_configurations[0].components[5].component = ARINC_PortManagerFactory_get_
↪interface();
    _system_configurations[0].components[5].listener = NULL;
    _system_configurations[0].components[5].property =
```

(continues on next page)

(continued from previous page)

```
        (struct RT_ComponentFactoryProperty*) _new_prtmgr_property();

    // ARINC 653 Transport
    if (!RTI_TSS_Micro_Component_initialize(&_system_configurations[0].components[6]))
    {
      return false;
    }
    _system_configurations[0].components[6].name = NETIO_DEFAULT_ARINC_NAME;
    _system_configurations[0].components[6].component = ARINC_InterfaceFactory_get_
↪interface();
    _system_configurations[0].components[6].listener = NULL;
    _system_configurations[0].components[6].property =
        (struct RT_ComponentFactoryProperty*) _new_arinc_property();

     ...
}

struct ARINC_PortManagerFactoryProperty *_new_prtmgr_property()
{
     struct ARINC_RouteProperty *route_property = NULL;
     struct ARINC_ReceivePortProperty *receive_port_prop = NULL;
     struct ARINC_PortManagerFactoryProperty *port_mgr_prop =
    (struct ARINC_PortManagerFactoryProperty *) malloc(
        sizeof(struct ARINC_PortManagerFactoryProperty));
    memset(port_mgr_prop, 0, sizeof(struct ARINC_PortManagerFactoryProperty));
    *port_mgr_prop = ARINC_PORTMANAGER_FACTORY_PROPERTY_DEFAULT;

    /* Configure synchronous or asynchronous processing.
     * Synchronous (TRUE) will use this Port Manager's receive thread to pass
     * a received message all the way up to the receiving application.
     * Asynchronous (FALSE) will enqueue the message with the ARINC Interface, which
     * then uses its receive thread to pass each message up to the receiving␣
↪application.
     */
     port_mgr_prop->sync_processing = RTI_TRUE;

    /* Configure APEX process attributes of the Port Manager's receive thread */
    port_mgr_prop->thread_property.port_property.parent.deadline = SOFT;
    port_mgr_prop->thread_property.port_property.parent.period = 100000000;
    port_mgr_prop->thread_property.port_property.parent.time_capacity = 100000000;
    port_mgr_prop->thread_property.stack_size = 32000;
    port_mgr_prop->thread_property.priority = 99;

    #define RTI_PORTMGR_RECV_THREAD_NAME "rti.ARINC.pm.rcv_thrd"
    OSAPI_Memory_zero(&port_mgr_prop->thread_property.port_property.parent.name, MAX_
↪NAME_LENGTH);
    OSAPI_Memory_copy(&port_mgr_prop->thread_property.port_property.parent.name,
        RTI_PORTMGR_RECV_THREAD_NAME,
        OSAPI_String_length(RTI_PORTMGR_RECV_THREAD_NAME));


    /* Configure send routes */
    ARINC_RoutePropertySeq_set_maximum(&port_mgr_prop->send_routes, 1);
    ARINC_RoutePropertySeq_set_length(&port_mgr_prop->send_routes, 1);
    route_property = ARINC_RoutePropertySeq_get_reference(
        &port_mgr_prop->send_routes, 0);
```

(continues on next page)

(continued from previous page)

```
    /* Configure a route from an APEX queuing port sending via a channel
     * with a unique channel identifier.
     */
    #define RTI_APP_SEND_PORT_NAME "Part1_send"
    route_property->channel_identifier = 1;
    route_property->send_port_property.max_message_size = 1024;
    route_property->send_port_property.max_queue_size = 32;
    OSAPI_Memory_copy(route_property->send_port_property.port_name,
        RTI_APP_SEND_PORT_NAME,
        OSAPI_String_length(RTI_APP_SEND_PORT_NAME));

    /* Configure APEX queuing ports to receive */
    ARINC_ReceivePortPropertySeq_set_maximum(&port_mgr_prop->receive_ports,1);
    ARINC_ReceivePortPropertySeq_set_length(&port_mgr_prop->receive_ports,1);
    receive_port_prop = ARINC_ReceivePortPropertySeq_get_reference(
        &port_mgr_prop->receive_ports,0);

    /* Configure an APEX queuing port receiving via a channel with
     * a unique channel identifier. */
    #define RTI_APP_RECV_PORT_NAME "Part1_recv"
    receive_port_prop->channel_identifier = 2;
    receive_port_prop->receive_port_property.max_message_size = 1024;
    receive_port_prop->receive_port_property.max_queue_size = 32;
    receive_port_prop->max_receive_queue_size = 32;
    OSAPI_Memory_copy(receive_port_prop->receive_port_property.port_name,
        RTI_APP_RECV_PORT_NAME,
        OSAPI_String_length(RTI_APP_RECV_PORT_NAME));

    return port_mgr_prop;
}

struct ARINC_InterfaceFactoryProperty *_new_arinc_property()
{
    struct ARINC_InterfaceFactoryProperty *arinc_property =
        (struct ARINC_InterfaceFactoryProperty *)
            malloc(sizeof(struct ARINC_InterfaceFactoryProperty));
    *arinc_property = ARINC_INTERFACE_FACTORY_PROPERTY_DEFAULT;

    /* Configure APEX process attributes of the ARINC Interface's receive thread.
     * Used only when this interface's Port Manager's sync_processing property is␣
↪FALSE.
     */
    arinc_property->recv_thread.port_property.parent.deadline = SOFT;
    arinc_property->recv_thread.port_property.parent.period = 100000000;
    arinc_property->recv_thread.port_property.parent.time_capacity = 100000000;
    arinc_property->recv_thread.stack_size = 16000;
    arinc_property->recv_thread.priority = 97;

    #define RTI_RECV_THREAD_NAME "rti.ARINC.p2.rcv_thrd"
    OSAPI_Memory_zero(&arinc_property->recv_thread.port_property.parent.name, MAX_
↪NAME_LENGTH);
    OSAPI_Memory_copy(&arinc_property->recv_thread.port_property.parent.name,
                      RTI_RECV_THREAD_NAME,
    OSAPI_String_length(RTI_RECV_THREAD_NAME));
```

(continues on next page)

(continued from previous page)

```
    return arinc_property;
}
```

DDS DomainParticipant configuration

The *DomainParticipant* configuration sets the properties and QoS of a DDS *DomainParticipant*. Each Transport Protocol Module connection relies on DDS entities created within a *DomainParticipant*, so each connection configuration itself will use a *DomainParticipant* configuration (`RTI_TSS_DDS_DomainParticipant_Configuration_T.config_name`).

Different *DomainParticipant* configurations can be created to account for any combination of different domain IDs or *DomainParticipant* QoS used by your TSS installation.

A *DomainParticipant* configuration is represented by `RTI_TSS_DDS_DomainParticipant_Configuration_T`.

Example configuration in C:

```
typedef struct RTI_TSS_DDS_DomainParticipant_Configuration
{
    /*name of the Domain Participant Configuration*/
    const char *config_name;

    /*domain_id that the Domain Participant belongs to*/
    DDS_DomainId_t domain_id;

    /*the configure_domain_participant_qos_fn that configures the domain participant␣
↪qos settings*/
    RTI_TSS_DomainParticipantQosCallback configure_domain_participant_qos_fn;

    /*additional configuration data */
    void *user_data;

    /*pointer to the domain participant listener */
    struct DDS_DomainParticipantListener dp_listener;

    /*listener status mask*/
    DDS_StatusMask dp_listener_status;

#if using Micro
    /*a structure describing the discovery configuration */
    struct RTI_TSS_DiscoveryConfigPlugin discovery_config;
#endif
#if using Pro
    /*a pointer to the qos_library for the Domain Participant*/
    const char *qos_library;

    /*a pointer to the qos_profile for the Domain Participant */
    const char *qos_profile;
#endif
} RTI_TSS_DDS_DomainParticipant_Configuration_T;
```

A note about *DomainParticipants* and *Topics*:

- Transport Protocol Module will only create a single *DomainParticipant* for each unique domain. If more than one connection uses the same domain, the *DomainParticipant* will be reused. A shared *DomainParticipant* will not be deleted until the last connection using the *DomainParticipant* is deleted.
- Similarly, Transport Protocol Module will create a single *Topic* for each unique domain. Much like the

DomainParticipant, the *Topic* will be reused if more than one connection uses the same *Topic* (in the same domain). A shared *Topic* will not be deleted until the last connection using the *Topic* is deleted (as a result of destroying the connection).

Note: It is important to know and configure the *DomainParticipant* QoS of the first created connection. Because only one *DomainParticipant* is created for each unique DDS domain, the corresponding participant QoS function of the first created connection will be applied.

Connext Micro DPSE configuration

Configuration of Dynamic-Participant, Static-Endpoint (DPSE) discovery for *Connext Micro* requires functions to be called to assert discoverable remote DDS endpoints. To support this, the *DomainParticipant* configuration has a discovery plugin field (`RTI_TSS_DDS_DomainParticipant_Configuration_T.discovery_plugin`) to configure the discoverable remote *Entities* for that *DomainParticipant*.

The following snippets configure DPSE to discover two remote *DomainParticipants*, one with a remote publisher and the other with a remote subscriber.

Example DPSE configuration snippet in C++:

```
/* Example DPSE snippet from a Foo_TSSConfigInterface.hpp */

FACE::Boolean _populate_domain_config()
{
    if (!RTI_TSS_DDS_DomainParticipant_Configuration_initialize(&_domain_
↪configurations[0]))
    {
        return false;
    }
    _domain_configurations[0].config_name = "publisher_domain";
    ...

#ifdef RTI_CONNEXT_MICRO
    _set_discovery_config(_domain_configurations[0].discovery_config);
    ...
#endif
    ...
}


#ifdef RTI_CONNEXT_MICRO
void _set_discovery_config(struct RTI_TSS_DiscoveryConfigPlugin &cfg)
{
    /* This configuration will be used by those participants using DPSE */
    FACE::UnsignedLong i;
    RTI_TSS_RemoteParticipantEntry_T *current_par_entry = NULL;
    RTI_TSS_RemoteSubscriptionEntry_T *current_sub_entry = NULL;
    RTI_TSS_RemotePublicationEntry_T *current_pub_entry = NULL;
    struct DDS_SubscriptionBuiltinTopicData rem_subscription_data =
        DDS_SubscriptionBuiltinTopicData_INITIALIZER;
    struct DDS_PublicationBuiltinTopicData rem_publication_data =
        DDS_PublicationBuiltinTopicData_INITIALIZER;

    RTI_TSS_DiscoveryConfigPlugin_initialize(&cfg);

    cfg.dpse_component_name = "dpse";
```

(continues on next page)

(continued from previous page)

```
    /* The amount of remote participant_entries that will be asserted */
    cfg.length = 2;
    cfg.participant_entries = (RTI_TSS_RemoteParticipantEntry_T *)
        malloc(sizeof(RTI_TSS_RemoteParticipantEntry_T) * cfg.length);
    memset(cfg.participant_entries, 0, sizeof(RTI_TSS_RemoteParticipantEntry_T));

    /* Configure the remote publisher participant */
    current_par_entry = &cfg.participant_entries[1];
    current_par_entry->participant_name = "publisher";

    current_par_entry->sub_length = 0;

    /* Configure remote publisher */
    current_par_entry->pub_length = 1;
    current_par_entry->publication_entries = (RTI_TSS_RemotePublicationEntry_T *)
        malloc( sizeof(RTI_TSS_RemotePublicationEntry_T) * current_par_entry->pub_
↪length);
    memset(current_par_entry->publication_entries, 0,
        sizeof(RTI_TSS_RemotePublicationEntry_T) * current_par_entry->pub_length);

    rem_publication_data.key.value[DDS_BUILTIN_TOPIC_KEY_OBJECT_ID] = 100;
    rem_publication_data.topic_name = DDS_String_dup("Example FACE_DM_RTI_
↪InteropExample");
    rem_publication_data.type_name = DDS_String_dup("FACE::DM::RTI::InteropExample");
    rem_publication_data.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;

    current_pub_entry = &current_par_entry->publication_entries[0];
    current_pub_entry->data = rem_publication_data;
    current_pub_entry->key_kind = FACE_DM_RTI_InteropExample_get_key_kind(
        FACE_DM_RTI_InteropExampleTypePlugin_get(), NULL);

    /* Configure the remote subscriber participant */
    current_par_entry = &cfg.participant_entries[0];
    current_par_entry->participant_name = "subscriber";

    current_par_entry->pub_length = 0;

    /* Configure remote subscriber */
    current_par_entry->sub_length = 1;
    current_par_entry->subscription_entries = (RTI_TSS_RemoteSubscriptionEntry_T *)
        malloc(sizeof(RTI_TSS_RemoteSubscriptionEntry_T) * current_par_entry->sub_
↪length);
    memset(current_par_entry->subscription_entries, 0,
        sizeof(RTI_TSS_RemoteSubscriptionEntry_T) * current_par_entry->sub_length);

    rem_subscription_data.key.value[DDS_BUILTIN_TOPIC_KEY_OBJECT_ID] = 200;
    rem_subscription_data.topic_name = DDS_String_dup("Example FACE_DM_RTI_
↪InteropExample");
    rem_subscription_data.type_name = DDS_String_dup("FACE::DM::RTI::InteropExample");
    rem_subscription_data.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;

    current_sub_entry = &current_par_entry->subscription_entries[0];
    current_sub_entry->data = rem_subscription_data;
    current_sub_entry->key_kind = FACE_DM_RTI_InteropExample_get_key_kind(
        FACE_DM_RTI_InteropExampleTypePlugin_get(), NULL);
```

(continues on next page)

(continued from previous page)

```
}
#endif //RTI_CONNEXT_MICRO
```

Example DPSE configuration snippet in C:

```
/* Example DPSE snippet from a Foo_TSSConfigInterface.hpp */

FACE_boolean _populate_domain_config()
{
    if (!RTI_TSS_DDS_DomainParticipant_Configuration_initialize(&_domain_
↪configurations[0]))
    {
        return false;
    }
    _domain_configurations[0].config_name = "publisher_domain";
    ...

#ifdef RTI_CONNEXT_MICRO
    _set_discovery_config(_domain_configurations[0].discovery_config);
    ...
#endif
    ...
}


#ifdef RTI_CONNEXT_MICRO
void _set_discovery_config(struct RTI_TSS_DiscoveryConfigPlugin &cfg)
{
    /* This configuration will be used by those participants using DPSE */
    FACE_unsigned_long i;
    RTI_TSS_RemoteParticipantEntry_T *current_par_entry = NULL;
    RTI_TSS_RemoteSubscriptionEntry_T *current_sub_entry = NULL;
    RTI_TSS_RemotePublicationEntry_T *current_pub_entry = NULL;
    struct DDS_SubscriptionBuiltinTopicData rem_subscription_data =
        DDS_SubscriptionBuiltinTopicData_INITIALIZER;
    struct DDS_PublicationBuiltinTopicData rem_publication_data =
        DDS_PublicationBuiltinTopicData_INITIALIZER;

    RTI_TSS_DiscoveryConfigPlugin_initialize(&cfg);

    cfg.dpse_component_name = "dpse";

    /* The amount of remote participant_entries that will be asserted */
    cfg.length = 2;
    cfg.participant_entries = (RTI_TSS_RemoteParticipantEntry_T *)
        malloc(sizeof(RTI_TSS_RemoteParticipantEntry_T) * cfg.length);
    memset(cfg.participant_entries, 0, sizeof(RTI_TSS_RemoteParticipantEntry_T));

    /* Configure the remote publisher participant */
    current_par_entry = &cfg.participant_entries[1];
    current_par_entry->participant_name = "publisher";

    current_par_entry->sub_length = 0;

    /* Configure remote publisher */
```

(continues on next page)

(continued from previous page)

```
    current_par_entry->pub_length = 1;
    current_par_entry->publication_entries = (RTI_TSS_RemotePublicationEntry_T *)
        malloc( sizeof(RTI_TSS_RemotePublicationEntry_T) * current_par_entry->pub_
↪length);
    memset(current_par_entry->publication_entries, 0,
        sizeof(RTI_TSS_RemotePublicationEntry_T) * current_par_entry->pub_length);

    rem_publication_data.key.value[DDS_BUILTIN_TOPIC_KEY_OBJECT_ID] = 100;
    rem_publication_data.topic_name = DDS_String_dup("Example FACE_DM_RTI_
↪InteropExample");
    rem_publication_data.type_name = DDS_String_dup("FACE::DM::RTI::InteropExample");
    rem_publication_data.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;

    current_pub_entry = &current_par_entry->publication_entries[0];
    current_pub_entry->data = rem_publication_data;
    current_pub_entry->key_kind = FACE_DM_RTI_InteropExample_get_key_kind(
        FACE_DM_RTI_InteropExampleTypePlugin_get(), NULL);

    /* Configure the remote subscriber participant */
    current_par_entry = &cfg.participant_entries[0];
    current_par_entry->participant_name = "subscriber";

    current_par_entry->pub_length = 0;

    /* Configure remote subscriber */
    current_par_entry->sub_length = 1;
    current_par_entry->subscription_entries = (RTI_TSS_RemoteSubscriptionEntry_T *)
        malloc(sizeof(RTI_TSS_RemoteSubscriptionEntry_T) * current_par_entry->sub_
↪length);
    memset(current_par_entry->subscription_entries, 0,
        sizeof(RTI_TSS_RemoteSubscriptionEntry_T) * current_par_entry->sub_length);

    rem_subscription_data.key.value[DDS_BUILTIN_TOPIC_KEY_OBJECT_ID] = 200;
    rem_subscription_data.topic_name = DDS_String_dup("Example FACE_DM_RTI_
↪InteropExample");
    rem_subscription_data.type_name = DDS_String_dup("FACE::DM::RTI::InteropExample");
    rem_subscription_data.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;

    current_sub_entry = &current_par_entry->subscription_entries[0];
    current_sub_entry->data = rem_subscription_data;
    current_sub_entry->key_kind = FACE_DM_RTI_InteropExample_get_key_kind(
        FACE_DM_RTI_InteropExampleTypePlugin_get(), NULL);

}
#endif //RTI_CONNEXT_MICRO
```

Type Support configuration

The Type Support configuration sets the DDS type-specific functions to be used for a specific type identified by a unique name (`RTI_TSS_TypeSupport_Configuration.type_name`).

Each Type Support configuration holds a pointer to the DDS `register_type` function for that type (`RTI_TSS_TypeSupport_Configuration.register_type_fn`).

A Type Support configuration is represented by `RTI_TSS_TypeSupport_Configuration_T`.

```
typedef struct RTI_TSS_TypeSupport_Configuration
{
    /*name of the type*/
    const char* type_name;

    /*the register_type_fn that registers the type*/
    RTI_TSS_RegisterTypeFunction register_type_fn;
} RTI_TSS_TypeSupport_Configuration_T;
```

The following snippets set up a Type Support configuration.

Example in C++:

```
/***** FACE::DM::RTI::HelloWorld *****/
if (!RTI_TSS_TypeSupport_Configuration_initialize(&_type_support_configurations[0]))
{
    return false;
}
_type_support_configurations[0].type_name = "FACE::DM::RTI::HelloWorld";
_type_support_configurations[0].register_type_fn =
        FACE_DM_RTI_HelloWorldTypeSupport_register_type;
```

Example in C:

```
/***** FACE_DM_RTI_HelloWorld *****/
if (!RTI_TSS_TypeSupport_Configuration_initialize(&_type_support_configurations[0]))
{
    return false;
}
_type_support_configurations[0].type_name = "FACE::DM::RTI::HelloWorld";
_type_support_configurations[0].register_type_fn =
        FACE_DM_RTI_HelloWorldTypeSupport_register_type;
```

Connection configuration

The Connection configuration sets a named connection and its underlying DDS *Entities*. The `RTI_TSS_Channel_Configuration_t.config_name` must match the connection_name parameter of `FACE::TSS::TPM::TPMTS::Open_Channel` (if using C++) or `FACE_TSS_TPM_TPMTS_Open_Channel` (if using C).

The connection configuration will use the Type Support configuration identified by the `RTI_TSS_Channel_Configuration_t.type_name` field.

Each connection's DDS *Entities* are managed by the *DomainParticipant* corresponding to the named *DomainParticipant* configuration (`RTI_TSS_Channel_Configuration_t.domain_config_name`). The DDS QoS for each DDS *Entity* type (*Publisher*, *Subscriber*, *DataWriter*, *DataReader*, *Topic*) are set by separate functions (`configure_*_qos_fnc()`).

A Connection configuration is represented by `RTI_TSS_Channel_Configuration_T`.

Example Connection configuration in C:

```
typedef struct RTI_TSS_Channel_Configuration
{
    /*name of the connection configuration */
    const char *config_name;

    /*name of the domain configuration*/
    const char *domain_config_name;

    /*direction indicates if the connection is a send or receive*/
    FACE_CONNECTION_DIRECTION_TYPE direction;

    /*name of the topic for the connection */
    const char *topic_name;

    /*name of the type for the connection */
    const char *type_name;

    /*the configure_publisher_qos_fn that configures the publisher qos settings */
    RTI_TSS_PublisherQosCallback configure_publisher_qos_fn;

    /*the publisher listener for this connection*/
    struct DDS_PublisherListener publisher_listener;

    /*the statusmask for the publisher listener */
    DDS_StatusMask publisher_listener_status;

    /*the configure_subscriber_qos_fn that configures the subscriber qos settings  */
    RTI_TSS_SubscriberQosCallback configure_subscriber_qos_fn;

    /*the subscriber listener for this connection*/
    struct DDS_SubscriberListener subscriber_listener;

    /*the statusmask for the subscriber listener  */
    DDS_StatusMask subscriber_listener_status;

    /*the configure_topic_qos_fn that configures the topic qos settings  */
    RTI_TSS_TopicQosCallback configure_topic_qos_fn;

    /*the topic listener for this connection */
    struct DDS_TopicListener topic_listener;

    /*the statusmask for the topic listener */
    DDS_StatusMask topic_listener_status;

#if using Pro
    /*name of the content filtered topic*/
    const char *content_filter_topic_name;
    /*the content filter expressionson*/
    const char *content_filter_topic_expr;
    /*the content filter parameters*/
    struct DDS_StringSeq content_filter_topic_params;
#endif

    /*the configure_datawriter_qos_fn that configures the datawriter qos settings  */
    RTI_TSS_DataWriterQosCallback configure_datawriter_qos_fn;

    /*the writer listener for this connection */
```

(continues on next page)

(continued from previous page)

```
    struct DDS_DataWriterListener writer_listener;

    /*the statusmask for the topic listener */
    DDS_StatusMask writer_listener_status;

    /*the configure_datareader_qos_fn that configures the datareader qos settings  */
    RTI_TSS_DataReaderQosCallback configure_datareader_qos_fn;

    /*the reader listener for this connection */
    struct DDS_DataReaderListener reader_listener;

    /*the statusmask for the topic listener */
    DDS_StatusMask reader_listener_status;

    /*additional configuration data */
    void *user_data;

#if using Pro
    /*set the connection to ignore itself */
    DDS_Boolean bi_dir_ignore_self;

    /*pointer to the qos_library for the connection*/
    const char* qos_library;

    /*pointer to the qos_profile for the connection*/
    const char* qos_profile;
#endif
} RTI_TSS_Channel_Configuration_T;
```

The following snippets configure one connection.

Example configuration in C++:

```
 /* Example Connection snippet from a Foo_TSSConfigInterface.hpp */

static const FACE::UnsignedLong _connection_configurations_length = 2 * 3;
RTI_TSS_Channel_Configuration_T _connection_configurations[_connection_configurations_
↪length];

FACE::Boolean _populate_connection_config()
{
    /***** FACE::DM::RTI::HelloWorld *****/
    if (!RTI_TSS_Channel_Configuration_initialize(&_connection_configurations[0]))
    {
        return false;
    }
    _connection_configurations[0].config_name = "FACE::DM::RTI::HelloWorld publisher";
    _connection_configurations[0].domain_config_name = "domain_0";
    _connection_configurations[0].direction = FACE_SOURCE;
    _connection_configurations[0].topic_name = "Example FACE::DM::RTI::HelloWorld";
    _connection_configurations[0].type_name = "FACE::DM::RTI::HelloWorld";
    _connection_configurations[0].configure_publisher_qos_fn = FACE_DM_RTI_HelloWorld_
↪publisher_qos;
    _connection_configurations[0].configure_topic_qos_fn = FACE_DM_RTI_HelloWorld_
↪topic_qos;
    _connection_configurations[0].configure_datawriter_qos_fn = FACE_DM_RTI_
↪HelloWorld_datawriter_qos;
```

(continues on next page)

(continued from previous page)

```
    _connection_configurations[0].configure_subscriber_qos_fn = NULL;
    _connection_configurations[0].configure_datareader_qos_fn = NULL;
    _connection_configurations[0].user_data = NULL;

#ifndef RTI_CONNEXT_MICRO
    _connection_configurations[0].bi_dir_ignore_self = DDS_BOOLEAN_TRUE;
    _connection_configurations[0].qos_library = "FACE_DM_RTI_HelloWorld_Library";
    _connection_configurations[0].qos_profile = "FACE_DM_RTI_HelloWorld_Profile";
#endif
    ...
}
```

Example configuration in C:

```
 /* Example Connection snippet from a Foo_TSSConfigInterface.hpp */

static const FACE_unsigned_long _connection_configurations_length = 2 * 3;
RTI_TSS_Channel_Configuration_T _connection_configurations[_connection_configurations_
↪length];

FACE_boolean _populate_connection_config()
{
    /***** FACE_DM_RTI_HelloWorld *****/
    if (!RTI_TSS_Channel_Configuration_initialize(&_connection_configurations[0]))
    {
        return false;
    }
    _connection_configurations[0].config_name = "FACE::DM::RTI::HelloWorld publisher";
    _connection_configurations[0].domain_config_name = "domain_0";
    _connection_configurations[0].direction = FACE_SOURCE;
    _connection_configurations[0].topic_name = "Example FACE::DM::RTI::HelloWorld";
    _connection_configurations[0].type_name = "FACE::DM::RTI::HelloWorld";
    _connection_configurations[0].configure_publisher_qos_fn = FACE_DM_RTI_HelloWorld_
↪publisher_qos;
    _connection_configurations[0].configure_topic_qos_fn = FACE_DM_RTI_HelloWorld_
↪topic_qos;
    _connection_configurations[0].configure_datawriter_qos_fn = FACE_DM_RTI_
↪HelloWorld_datawriter_qos;
    _connection_configurations[0].configure_subscriber_qos_fn = NULL;
    _connection_configurations[0].configure_datareader_qos_fn = NULL;
    _connection_configurations[0].user_data = NULL;

#ifndef RTI_CONNEXT_MICRO
    _connection_configurations[0].bi_dir_ignore_self = DDS_BOOLEAN_TRUE;
    _connection_configurations[0].qos_library = "FACE_DM_RTI_HelloWorld_Library";
    _connection_configurations[0].qos_profile = "FACE_DM_RTI_HelloWorld_Profile";
#endif
    ...
}
```

Note: Per the FACE TSS specification, the QoS for a connection's *Entities* is established at creation time for that connection and cannot be dynamically modified during communication.

Note: If a bi-directional connection is created (with `bi_dir_ignore_self` set to `TRUE`), it will not be able to communicate with other connections in the same instance with the same *DomainParticipant* configuration.

This is because DDS_DomainParticipant_ignore_publication() is called internally when creating a new connection with bi_dir_ignore_self set to TRUE. For more information, refer to the API Reference in the *Connext Micro* User's Manual located in your *Connext Micro* installation directory (<RTIMEHOME>\doc).

6.7.5 How configuration data is used in C++

An RTI::Configuration instance is set within a FACE::TSS::TPM::TPMTS_impl instance by calling the FACE::TSS::TPM::TPMTS_impl::Set_Reference() function. A subsequent call to FACE::TSS::TPM::TPMTS_impl::Initialize() will initialize the TPM instance with the configuration data in RTI::Configuration. If this is the first call to FACE::TSS::TPM::TPMTS_impl::Initialize(), the system property will be used to allocate the resources for Transport Protocol Module and DDS.

When creating a connection, the connection configuration name (RTI_TSS_Channel_Configuration_t.config_name) matching the connection_name parameter to FACE::TSS::TPM::TPMTS_impl::Open_Channel() is used to configure the created connection. Open_Channel() will fail and return a FACE::RETURN_CODE_TYPE::Value::INVALID_CONFIG``if no matching ``config_name is found.

6.7.6 How configuration data is used in C

An RTI_Configuration instance is set within an RTI_TPM instance by calling Configuration_Injectable_Set_Reference() with an RTI_TPM cast as a FACE_Configuration_Injectable *. A subsequent call to FACE_TSS_TPM_TPMTS_Initialize() initializes the TPM instance with the configuration data in RTI_Configuration. If this is the first call to FACE_TSS_TPM_TPMTS_Initialize(), the system property will be used to allocate the resources for Transport Protocol Module and DDS.

When creating a connection, the connection configuration name (RTI_TSS_Channel_Configuration_t.config_name) matching the connection_name parameter to FACE_TSS_TPM_TPMTS_Open_Channel() is used to configure the created connection. Open_Channel() will fail and return a FACE_RETURN_CODE_TYPE_INVALID_CONFIG error if no matching config_name is found.

6.7.7 Extending the configuration interface

You are required to implement the configuration to suit your needs. You can copy the HelloWorld example configuration for your language as a starting point and working reference. The only hard requisites are:

- All FACE Configuration functions must be implemented.
- In a C FACE Configuration instance, the following variables must be defined in the Configuration Implementation provided to FACE_TSS_Base:

```
/*ce \dref_RTI_TSS_SYSTEM_CONFIGURATION_CONTAINER_NAME */
FACE_string RTI_TSS_SYSTEM_CONFIGURATION_CONTAINER_NAME;

/*ce \dref_RTI_TSS_DOMAIN_PARTICIPANT_CONFIGURATION_CONTAINER_NAME */
FACE_string RTI_TSS_DOMAIN_PARTICIPANT_CONFIGURATION_CONTAINER_NAME;

/*ce \dref_RTI_TSS_TYPE_SUPPORT_CONFIGURATION_CONTAINER_NAME */
FACE_string RTI_TSS_TYPE_SUPPORT_CONFIGURATION_CONTAINER_NAME;

/*ce \dref_RTI_TSS_CONNECTION_CONFIGURATION_CONTAINER_NAME */
FACE_string RTI_TSS_CHANNEL_CONFIGURATION_CONTAINER_NAME;
```

The above variables must also be initialized as follows:

```
FACE_string_init_managed_cstring(&RTI_TSS_SYSTEM_CONFIGURATION_CONTAINER_NAME,
↪"RTI_TSS_SYSTEM_CONFIGURATION");
FACE_string_init_managed_cstring(&RTI_TSS_DOMAIN_PARTICIPANT_CONFIGURATION_
↪CONTAINER_NAME, "RTI_TSS_DOMAIN_PARTICIPANT_CONFIGURATION");
FACE_string_init_managed_cstring(&RTI_TSS_TYPE_SUPPORT_CONFIGURATION_CONTAINER_
↪NAME, "RTI_TSS_TYPE_SUPPORT_CONFIGURATION");
FACE_string_init_managed_cstring(&RTI_TSS_CHANNEL_CONFIGURATION_CONTAINER_NAME,
↪"RTI_TSS_CONNECTION_CONFIGURATION");
```

- The `Open` method should return a handle for each one of the well-known container names defined at:
  - `RTI_TSS_SYSTEM_CONFIGURATION_CONTAINER_NAME`
  - `RTI_TSS_DOMAIN_PARTICIPANT_CONFIGURATION_CONTAINER_NAME`
  - `RTI_TSS_TYPE_SUPPORT_CONFIGURATION_CONTAINER_NAME`
  - `RTI_TSS_CHANNEL_CONFIGURATION_CONTAINER_NAME`
- The `Read` method should return, based on the handle returned by `Open`, a reference to an object of type:
  - `RTI_TSS_System_Configuration_T`
  - `RTI_TSS_DDS_DomainParticipant_Configuration_T`
  - `RTI_TSS_TypeSupport_Configuration_T`
  - `RTI_TSS_Channel_Configuration_t`

6.8 Configuring FACE 3.x for Connext Professional

6.8.1 Connext Professional data types

The *Connext Professional* Transport Protocol Module uses XML-based application creation to handle configuration. Refer to the XML-Based Application Creation Getting Started Guide and Configuring QoS with XML for more information on this process.

There is an external hash table in `rti_connext_tss_<version>/code/C/include/pro/Configuration_Pro.h` that contains the list of types and type sizes that *Connext Professional* will use during the lifetime of the application.

6.8.2 Registering data types

In a C FACE Configuration instance, you must register all types to be transmitted in a message with the `DomainParticipantFactory` in the `configuration_initialization` call, as shown in the following code snippet:

```
DDS_ReturnCode_t retCode = DDS_DomainParticipantFactory_register_type_support(
        DDS_TheParticipantFactory,
        FACE_DM_HelloWorldTypeSupport_register_type,
        "FACE::DM::RTI::HelloWorld");
    if (retCode != DDS_RETCODE_OK) {
        // Handle error
    }
```

You must also register all types and their sizes with the types hash table in the `configuration_initialization` call, as shown below:

```
ht_result = types_hash_table_add_entry("FACE::DM::RTI::HelloWorld", sizeof(FACE_DM_
↪HelloWorld)); /* Replace with actual size */
if (ht_result != HT_NO_ERROR)
{
    *return_code = FACE_RETURN_CODE_TYPE_RESOURCE_LIMIT_REACHED;
    return FACE_INTERFACE_INSUFFICIENT_MEMORY;
}
```

When reading a configuration, the string passed in must be matched to a *DataReader/DataWriter* configuration (found in XML app creation) passed into the system as a reference.

The `configuration_read` call acts as a lookup table that returns the fully-qualified DDS entity name (from XML QoS) for a given connection name. The FACE TSS uses these names to locate and create DDS *DomainParticipants*, *Publishers*, *Subscribers*, *DataReaders*, and *DataWriters*.

For Connext Pro integrations using XML Application Creation, these entities must be auto-created from the XML configuration before channel I/O starts.

6.8.3 Adding new participants

1. Add a new `if()` block checking for your *DomainParticipant* name:

```
if(0 == strcmp(FACE_string_buffer(set_name), "your_participant_name"))
```

2. Clear the buffer and copy the *DomainParticipant*'s fully-qualified name from your XML:

```
*bytes_read = 0;
memset(buffer, '\0', buffer_size);
strcat(buffer, "YourLibraryName::YourParticipantName\0");
*bytes_read += strlen(buffer);
```

3. Add `goto done;` to complete the lookup.

6.8.4 Adding new DataReaders and DataWriters

1. Add a new `if()` block checking for your connection name:

```
if(0 == strcmp(FACE_string_buffer(set_name), "your_connection_name"))
```

2. Copy the full path to the *DataReader* or *DataWriter* from your XML QoS:

```
*bytes_read = 0;
memset(buffer, '\0', buffer_size);
strcat(buffer, "YourParticipant::YourSubscriber::YourDataReader\0");
*bytes_read += strlen(buffer);
```

3. Add `goto done;`.

6.8.5 Example

To add a new "Temperature" data type with Publish/Subscribe *DomainParticipants* and a *DataReader*:

```
if(0 == strcmp(FACE_string_buffer(set_name), "temp_pub_participant"))
{
    *bytes_read = 0;
    memset(buffer, '\0', buffer_size);
    strcat(buffer, "TempLibrary::TempPublisher\0");
    *bytes_read += strlen(buffer);

    goto done;
}

if(0 == strcmp(FACE_string_buffer(set_name), "temp_receive"))
{
    *bytes_read = 0;
    memset(buffer, '\0', buffer_size);
    strcat(buffer, "TempLibrary::TempSubscriber::TempReader\0");
    *bytes_read += strlen(buffer);
    goto done;
}
```

Note: Connection names (`set_name` in the code snippets above) must match what your application passes to `FACE_TSS_Create_Connection()`. Entity names must match your XML QoS profile exactly.

Note: Currently, the *Connext Professional* Transport Protocol Module does not accept explicitly-defined XML app creation files (see Loading XML Configuration Files) and instead looks for a `USER_QOS_PROFILES.xml` in the application runtime area.

6.9 Dynamic Memory Allocation

Connext TSS does *not* free dynamically allocated memory when built for FACE Safety Base or Safety Extended profiles. Dynamically allocated memory in the generated TSS plugins is also *not* returned.

6.10 Sequence Element Lifecycle Operations

The FACE Technical Standard defines sequences as generic containers but does not prescribe a mechanism for sequences to perform deep cleanup or deep copy of their elements. This means that when a sequence contains elements of a complex type (such as strings or nested structures that allocate internal memory), freeing the sequence's buffer alone does not release the memory owned by individual elements. This results in memory leaks.

6.10.1 C++ API

The C++ `FACE::Sequence<T>` template class handles element lifecycle **automatically** through C++ language features. No additional configuration is required:

- **Destruction**: The sequence destructor calls each element's destructor via `FACE::internal::sequence_helper<T>::destroy_elements()`.
- **Deep copy**: Copy construction and assignment use `FACE::internal::sequence_helper<T>::copy_elements()`, which invokes each element's copy constructor via placement new.
- **Specializations**: `FACE::String`, nested `FACE::Sequence<T>`, primitive types, and fixed-size arrays each have their own `sequence_helper` specialization with appropriate behavior.

For custom C++ types used as sequence elements, simply ensure your type has a proper copy constructor and destructor. The `FACE::Sequence<T>` template will invoke them automatically.

6.10.2 C API

The C API does not have constructors or destructors, so *Connext TSS* provides optional **element lifecycle operations** on the `FACE_sequence` type. Each sequence instance has an `ops` field with two function pointers:

```
typedef void (*FACE_sequence_clean_elem_fn)(void *elem);
typedef void (*FACE_sequence_copy_elem_fn)(void *dst_elem,
                                           const void *src_elem,
                                           FACE_unsigned_long type_size);

typedef struct {
    FACE_sequence_copy_elem_fn  copy_elem_fn;
    FACE_sequence_clean_elem_fn clean_elem_fn;
} FACE_sequence_ops;
```

When set, the sequence implementation calls `clean_elem_fn` on each element during `FACE_sequence_free()` and `FACE_sequence_clear()`, and calls `copy_elem_fn` during `FACE_sequence_append()`, `FACE_sequence_append_elem()`, `FACE_sequence_init_managed_copy()`, and `FACE_sequence_init_managed_data()` to perform deep copies. When these function pointers are NULL (the default), sequences behave as before (shallow `memset`/`memcpy`).

6.10.3 Move Semantics in `FACE_sequence_append_elem`

When `copy_elem_fn` is set, `FACE_sequence_append_elem()` uses **move semantics**: after deep-copying the source element into the sequence buffer, it automatically calls `clean_elem_fn` on the source to release its internal resources. This ensures that:

- **Existing code** that does not free the source after appending will not leak memory — the source's resources are consumed by the append operation.
- **New code** that explicitly frees the source after appending remains safe — cleaning an already-cleaned element is a no-op.

```
FACE_sequence inner;
memset(&inner, 0, sizeof(FACE_sequence));
FACE_sequence_init_managed_bounded(&inner, sizeof(FACE_long), 10);
/* ... populate inner ... */

/* append_elem deep-copies inner into outer_seq, then cleans the source */
FACE_sequence_append_elem(&outer_seq, &inner, sizeof(FACE_sequence));

/* inner's internal buffer has been released - no leak even without
 * an explicit FACE_sequence_free(&inner) call. Calling free is still
 * safe but not required. */
```

Note: `FACE_sequence_append()` (which appends an entire sequence) does **not** apply move semantics to the source sequence — only individual element appends via `FACE_sequence_append_elem()` consume the source.

6.10.4 Enabling Sequence Element Operations

This feature is controlled by the `LIFECYCLE_CLASS_EXT` CMake option, which defaults to **ON**. When enabled, the `LIFECYCLE_CLASS_EXT` preprocessor macro is defined automatically during the build.

To explicitly disable this feature, pass `-DLIFECYCLE_CLASS_EXT=OFF` when configuring:

```
cmake -DLIFECYCLE_CLASS_EXT=OFF ...
```

When `LIFECYCLE_CLASS_EXT` is enabled (the default), the generated type plugin code (produced by *rtiddsgen*) automatically assigns the appropriate `clean_elem_fn` and `copy_elem_fn` for each sequence-of-user-type member during the type's `_initialize` function. No application-level code changes are required.

With `LIFECYCLE_CLASS_EXT` set to OFF, the generated code compiles identically to previous releases (no ops assigned, shallow behavior).

Warning: Disabling `LIFECYCLE_CLASS_EXT` restores shallow sequence element behavior for C sequence elements that own internal resources. In this mode, application-level memory leaks can occur unless your code explicitly performs equivalent deep cleanup and deep-copy handling.

6.10.5 RTI FACE_string Reference Implementation

Connext TSS provides a reference implementation of the element lifecycle operations for the `FACE_string` type in `FACE/string_ext.h` and `FACE/string_ext.c`:

```
#include "FACE/string_ext.h"

/* Releases all resources held by a FACE_string element */
void FACE_string_seq_clean_elem(void *elem);

/* Deep-copies a FACE_string element (initializes dst, copies content) */
void FACE_string_seq_copy_elem(void *dst_elem,
                               const void *src_elem,
                               FACE_unsigned_long type_size);
```

These are automatically assigned by the generated code when `LIFECYCLE_CLASS_EXT` is ON (the default). They serve as the canonical example for implementing element operations for any custom type.

6.10.6 Generated Element Operations

When `LIFECYCLE_CLASS_EXT` is ON, the *rtiddsgen* code generator automatically produces `<TypeName>_seq_clean_elem` and `<TypeName>_seq_copy_elem` functions for **every** struct, union, and typedef defined in your IDL. These generated functions call the type's `_finalize` and `_copy` functions respectively. No additional implementation is required for IDL-defined types.

The generated type plugin also assigns these functions to the `ops` field of each sequence member during the type's `_initialize` function.

6.10.7 Implementing Element Operations for Custom Types

If you have a hand-written C type (not generated from IDL) that holds internal resources (allocated memory, file handles, etc.) and that type appears as a sequence element, you **must** implement two functions following this exact naming convention:

```
void <TypeName>_seq_clean_elem(void *elem);
void <TypeName>_seq_copy_elem(void *dst_elem,
                              const void *src_elem,
                              FACE_unsigned_long type_size);
```

Where `<TypeName>` is the fully-qualified C type name with `::` replaced by `_` (matching the name used in the generated code). For example, if your IDL defines `module MyCompany { struct Widget { ... }; };`, the type name in C is `MyCompany_Widget` and the functions must be named `MyCompany_Widget_seq_clean_elem` and `MyCompany_Widget_seq_copy_elem`.

Requirements for `<TypeName>_seq_clean_elem`:

1. The function receives a `void*` pointer to a single element within the sequence buffer.
2. Cast the pointer to your concrete type: `MyType *typed = (MyType *)elem;`
3. If the pointer is not NULL, release all internal resources held by that element (free allocated memory, close handles, etc.).
4. Do **not** free the element pointer itself — it is part of the sequence's contiguous buffer.

Example:

```
void MyCompany_Widget_seq_clean_elem(void *elem)
{
    MyCompany_Widget *widget = (MyCompany_Widget *)elem;
    if (widget != NULL)
    {
        /* Release internal resources */
        free(widget->name_buffer);
        widget->name_buffer = NULL;
        widget->name_length = 0;
    }
}
```

Requirements for `<TypeName>_seq_copy_elem`:

1. The function receives a `void*` destination pointer, a `const void*` source pointer, and the element `type_size` (which may be ignored if your type is fixed-size or you already know it).
2. Cast both pointers to your concrete type.
3. **Finalize** the destination element first (it may contain live resources from a previous value that must be released to avoid leaks), then **initialize** it to a clean state.
4. Perform a **deep copy** of the source element's content into the initialized destination.

5. The destination may also be uninitialized memory (e.g., a freshly allocated buffer slot). Your implementation must handle both cases — finalize on uninitialized/zeroed memory should be a safe no-op.

Example:

```
void MyCompany_Widget_seq_copy_elem(
        void *dst_elem,
        const void *src_elem,
        FACE_unsigned_long type_size)
{
    (void)type_size;
    MyCompany_Widget *dst = (MyCompany_Widget *)dst_elem;
    const MyCompany_Widget *src = (const MyCompany_Widget *)src_elem;

    /* Finalize destination (releases any existing resources; safe on
     * zeroed/uninitialized memory since name_buffer will be NULL) */
    free(dst->name_buffer);

    /* Initialize destination to a clean state */
    memset(dst, 0, sizeof(MyCompany_Widget));

    /* Deep copy: allocate and copy the name buffer */
    if (src->name_buffer != NULL && src->name_length > 0)
    {
        dst->name_buffer = malloc(src->name_length + 1);
        if (dst->name_buffer != NULL)
        {
            memcpy(dst->name_buffer, src->name_buffer, src->name_length);
            dst->name_buffer[src->name_length] = '\0';
            dst->name_length = src->name_length;
        }
    }
    dst->id = src->id;
}
```

Linking: For hand-written (non-IDL) types, ensure that the object file or library containing your `_seq_clean_elem` and `_seq_copy_elem` implementations is linked into your final application. For IDL-defined types, these functions are generated automatically in the type plugin source file.

Summary of the approach:

- **The FACE specification does not define a mechanism for element-level resource** management within sequences. This is a known gap that leads to memory leaks for sequences of types that own dynamically allocated resources.
- ***Connext TSS* extends the sequence with optional ops function pointers that,** when set, enable proper deep cleanup and deep copy at the element level. This is a Connext TSS extension specifically implemented to address the above FACE specification shortcoming.
- **The `LIFECYCLE_CLASS_EXT` CMake option (defaulting to ON) controls this** extension. Setting it to OFF disables the feature and does not affect binary compatibility or compliance, but can reintroduce memory leak risk for sequence element types that allocate internal resources.
- For IDL-defined types, the generated type plugin code produces both the lifecycle functions and the ops assignments automatically — no manual implementation is required.
- For hand-written C types, implementors provide the two lifecycle functions following the documented naming convention.

7 Integrating a Third-Party TPM

This section describes how to integrate an existing Transport Protocol Module (TPM) implementation with the *Connext TSS* framework. It assumes you already have a working transport layer and need to wire it into the *Connext TSS* architecture.

7.1 Integration Steps

To integrate your third-party TPM with the *Connext TSS* framework, follow the subsections below in order.

7.1.1 Wrap your TPM in the FACE interface

Your TPM must expose the `FACE::TSS::TPM::TPMTS` interface. In C, embed the `FACE_TSS_TPM_TPMTS` struct and populate its function pointer table. In C++, inherit from the abstract class.

For example, see the following C struct layout:

```
typedef struct {
    FACE_TSS_TPM_TPMTS tpm;                                    // required
    FACE_Configuration_Injectable configuration_injectable;  // required
    FACE_Logging_Injectable logging_injectable;               // optional
    void *data;                                                // your private state
} MY_TPM;
```

And see the following example C++ class:

```
class MyTPM : public FACE::TSS::TPM::TPMTS,
              public FACE::Configuration_Injectable::Injectable {
    // override all TPMTS pure virtuals
    // override Set_Reference for Configuration
};
```

Next, wire the function pointer table in an initialization function (C only):

```
void MY_TPM_init(MY_TPM *this_obj) {
    this_obj->tpm.ops.Initialize = MY_TPM_Initialize_impl;
    this_obj->tpm.ops.Open_Channel = MY_TPM_Open_Channel_impl;
    this_obj->tpm.ops.Close_Channel = MY_TPM_Close_Channel_impl;
    this_obj->tpm.ops.Write_To_Transport = MY_TPM_Write_To_Transport_impl;
    this_obj->tpm.ops.Read_From_Transport = MY_TPM_Read_From_Transport_impl;
```

(continues on next page)

(continued from previous page)

```
    this_obj->tpm.ops.Register_TPM_Callback = MY_TPM_Register_TPM_Callback_impl;
    this_obj->tpm.ops.Unregister_TPM_Callback = MY_TPM_Unregister_TPM_Callback_impl;
    this_obj->tpm.ops.Is_Data_Available = MY_TPM_Is_Data_Available_impl;
    this_obj->tpm.ops.Get_TPM_Status = MY_TPM_Get_TPM_Status_impl;
    this_obj->tpm.ops.Request_TPM_State_Change = MY_TPM_Request_TPM_State_Change_impl;
    this_obj->configuration_injectable.ops.Configuration_Injectable_Set_Reference =
        MY_TPM_Configuration_Set_Reference_impl;
}
```

See *TSS TPM Interfaces* for complete parameter documentation.

7.1.2 Register your TPM with the application

TPMs are statically instantiated; there is no dynamic loading or plugin discovery. In your application startup code, instantiate your TPM with the following code:

```
// 1. Instantiate
MY_TPM my_tpm;
MY_TPM_init(&my_tpm);

// 2. Inject configuration service
FACE_Configuration_Injectable_Set_Reference(
    &my_tpm.configuration_injectable,
    &config_name, config_service, config_guid, &retcode);

// 3. Initialize your TPM
FACE_TSS_TPM_TPMTS_Initialize(&my_tpm.tpm, &tpm_config_name, &retcode);

// 4. Register with TSS Base (GUID must match TSS.xml)
FACE_TSS_TPM_TPMTS_Injectable_Set_Reference(
    &Base.tpm_injectable,
    &tpm_interface_name,
    &my_tpm,
    5,          // GUID matching your XML <tpm> element
    &retcode);
```

7.1.3 Declare your TPM in TSS.xml

Declare your TPM in the file `TSS.xml` with the following snippet:

```
<TSS>
  <configs>
    <config name='my_config'>
      <connections>
        <connection name='MyConnection'>
          <tpms><tpm guid='5'/></tpms>
        </connection>
      </connections>
      <tpms>
        <tpm name='MY_TPM' guid='5'/>
      </tpms>
    </config>
  </configs>
</TSS>
```

The `guid` attribute links your registered TPM instance to the connections that use it. This value must match the GUID passed in *Register your TPM with the application*.

7.2 Integration Constraints

This section describes the contracts your TPM must satisfy in order to work correctly with the *Connext TSS* Type Abstraction (TA) layer. Violating any of these contracts will cause silent data corruption, crashes, or deadlocks.

7.2.1 Data format contract

All data passes between layers as a `FACE_TSS_MESSAGE_TYPE`, as shown below:

```
typedef struct FACE_TSS_MESSAGE_TYPE {
  FACE_TSS_MESSAGE_GUID_TYPE message_guid;
  FACE_TSS_DATA_BUFFER_TYPE buffer;
} FACE_TSS_MESSAGE_TYPE;

typedef struct FACE_TSS_DATA_BUFFER_TYPE {
  FACE_SYSTEM_ADDRESS_TYPE buffer_address;   // void*
  FACE_TSS_BYTE_SIZE_TYPE buffer_capacity;   // size in bytes
} FACE_TSS_DATA_BUFFER_TYPE;
```

The TA layer passes this structure through unchanged; it never transforms or copies the data. Your TPM is the only layer that performs serialization or deserialization. Therefore, your TPM must satisfy the following contracts:

- **Sending** (`Write_To_Transport`): `buffer_address` points to the typed object in its in-memory C/C++ struct layout. Serialize it into your wire format.
- **Receiving** (callback or `Read_From_Transport`): Deserialize your wire data into the exact in-memory struct layout the TS layer expects.

> **Warning:** If your byte layout does not match (field order, padding, endianness, etc.), the application will receive corrupt data silently; there is no validation at the TA layer.

- **Buffer ownership on receive**: The TA layer passes your `buffer_address` directly to the TS layer without copying. Your buffer must remain valid after the callback returns; use heap memory or a pre-allocated pool.
- **GUID matching**: The `message_guid` you set on receive must match the GUID configured for the channel's type. The TA layer uses it to route messages.

`Write_To_Transport` and `Read_From_Transport` also pass a TSS header, as shown below:

```
typedef struct FACE_TSS_HEADER_TYPE {
  FACE_TSS_UID_TYPE instance_uid;
  FACE_TSS_UID_TYPE source_uid;
  FACE_SYSTEM_TIME_TYPE timestamp;
} FACE_TSS_HEADER_TYPE;
```

On sending, the TA layer provides these values. Transmit them if your transport carries per-message metadata. Upon receiving, populate from your transport's metadata or set to zero if unavailable.

7.2.2 Threading contract

The *Connext TSS* framework uses a synchronous, non-queued threading model. This section describes the threading model in detail.

Thread safety

Connext TSS protects the shared Type Abstraction or Base state (e.g., connection and callback metadata) with internal framework synchronization.

`Send_Message()` and `Receive_Message()` may run concurrently. No framework lock is held across Transport Protocol Module I/O, so transport work that blocks does not block other tasks.

The callback registration/unregistration state is synchronized at the framework level.

Sending data is synchronous through your entire `Write_To_Transport()` call. Receiving callbacks are also synchronous; your transport thread invokes the callback, and processing completes before control returns.

See *Thread Safety Model* in the *API Reference* for more information.

Application responsibilities

Your Transport Protocol Module and application *must* satisfy the following concurrency rules:

1. The Transport Protocol Module I/O must be thread-safe. `Write_To_Transport()` and `Read_From_Transport()` may be active concurrently on different threads.
2. Do not hold locks across callback invocation. If callback code calls back into `Send_Message()` (re-entering `Write_To_Transport()`), a held lock can cause a deadlock. Use separate send/recieve locks, or release locks before invoking callbacks.
3. After `Unregister_TPM_Callback()` returns, no further callbacks may fire. Ensure your receive path checks registration state and waits for in-flight callback completion as needed.
4. Keep initialization and teardown ordered: inject references first, then initialize, then unregister/close before teardown. Even with internal synchronization, this avoids lifecycle races in application code.

7.2.3 Callback contract

The following rules apply regarding callbacks:

- Only one active callback is allowed per channel. Return `NO_ACTION` on duplicate registration.
- No callbacks may fire after `Unregister_TPM_Callback()` returns. Your receiving thread must check the registration state before each invocation.

  This is an obligation on your Transport Protocol Module, and it is what *Connext TSS* relies on. It is not the same statement as the *Connext TSS*-level one at *FACE::TSS::BASE::Unregister_Callback()*, which says a final callback invocation may still be completing when `Unregister_Callback()` returns. *Connext TSS* covers the case where the underlying detach is not synchronous; your Transport Protocol Module must still stop dispatching once `Unregister_TPM_Callback()` has returned.
- Callbacks block the entire receive path; do not perform unbounded input/output (I/O) inside.

7.2.4 Error code contract

Your TPM must return these FACE codes for the following errors:

Error	Code
Timeout	`TIMED_OUT`
Invalid channel ID	`INVALID_PARAM`
Not initialized / unavailable	`NOT_AVAILABLE`
Buffer null or too small	`INVALID_PARAM`
Duplicate callback	`NO_ACTION`
Invalid configuration	`INVALID_PARAM`

7.2.5 SafetyBase profile constraints

The following constraints apply if your TPM targets the SafetyBase profile:

- No `free()` or `delete` functions are allowed at runtime. You must allocate everything during the `Initialize()` and `Open_Channel()` functions.
- No entity deletion. Once channels are open, do not destroy transport resources.
- No heap reallocation. Only bounded sequences and strings are allowed.
- Pre-allocate buffer pools during initialization for use during I/O.

7.3 Troubleshooting

This section describes problems you may encounter during or after integrating a third-party TPM, along with likely causes and fixes.

7.3.1 Connection returns NOT_AVAILABLE

Possible causes:

- `Initialize()` was called before injecting required services.
- The TPM was never registered with TSS Base. Verify that `FACE_TSS_TPM_TPMTS_Injectable_Set_Reference()` is called during startup.

Things to check:

1. Ensure all `Set_Reference` calls (such as Configuration, Serialization, or Logging) are complete before calling `Initialize()`.
2. Print or log the return code from each `Set_Reference` call. If any returns non-zero, the injectable was not accepted.
3. Confirm that the configuration service is fully populated (file loaded, entries parsed) before injection.
4. Add a log statement at the top of your `Initialize()` implementation to confirm it is being called at all.
5. Verify that `FACE_TSS_TPM_TPMTS_Injectable_Set_Reference()` is called during startup.

7.3.2 Send or receive silently produces corrupt data

Possible causes:

- The serialization byte layout does not match the in-memory struct layout the TS layer expects.
- On receive, `message_guid` is not set to the correct type GUID. The TA layer routes by this value; an incorrect GUID delivers data to the wrong handler or drops it.

Things to check:

1. Check field order, padding, alignment, and endianness.
2. Hex-dump the buffer contents immediately after serialization (on the sending side) and immediately after deserialization (on the receiving side). Compare byte-for-byte against the expected struct layout using `offsetof()` for each field.
3. Verify `sizeof(YourType)` matches on both sides. Compiler padding differences (e.g., `#pragma pack`) between your TPM and the generated type code are a common source of mismatch.
4. Log `message_guid` on both send/ and receive. Compare against the GUID value in `TSS.xml` for the connection's type.
5. If using a network transport, capture packets and verify wire-format bytes match your serialized output.

7.3.3 Callback never fires on receive

Possible causes:

- `message_guid` on the received message does not match the channel's configured type GUID. The TA layer silently discards unmatched messages.
- `Register_TPM_Callback()` was not called, or was called on the wrong channel.
- The TPM's receive thread is not running.

Things to check:

1. Add a log statement inside your receive thread loop to confirm it is executing and receiving data from the transport.
2. Log the `message_guid` you set on each received message and compare it against the GUID in your `TSS.xml` <connection> element.
3. Verify `Register_TPM_Callback()` returned `NO_ERROR`. If it returned `NO_ACTION`, a callback was already registered.
4. Confirm the channel ID passed to `Register_TPM_Callback()` matches the one returned by `Open_Channel()`.
5. Check that your receive thread was started *after* `Open_Channel()` completes, not during `Initialize()`.

7.3.4 Deadlock during send or receive

Possible causes:

- A single lock is held across both the send path and the callback invocation path. If the application's callback handler calls `Send_Message()` (re-entering `Write_To_Transport`), the lock will deadlock.

Things to check:

1. Use separate locks for send and receive, or release locks before invoking callbacks.
2. Attach a debugger and inspect all thread backtraces when the hang occurs. Look for two threads waiting on the same mutex.

3. Identify whether your callback handler (or anything it calls) invokes `Send_Message()` or any other TSS API that re-enters the TPM.
4. Temporarily replace your mutex with a try-lock and log failures. This reveals contention without hanging.
5. Verify that you do not hold a lock when invoking the registered callback function pointer.

7.3.5 Crash or use-after-free during shutdown

Possible causes:

- Callback fires after `Unregister_TPM_Callback()` returns. Your receive thread must check registration state before each callback invocation.
- Buffers are freed before the TS layer finishes processing. The TA layer passes `buffer_address` directly to the TS layer without copying.

Things to check:

1. Run under a memory sanitizer (ASan) or Valgrind to identify the exact access-after-free location.
2. Verify that `Unregister_TPM_Callback()` waits (e.g., via a condition variable or join) until any in-progress callback has returned before it completes.
3. Confirm that your receive-side buffers are not freed inside the callback itself. They must remain valid until the callback returns to your TPM code.
4. Add a flag that your receive thread checks *before* each callback invocation; set it in `Unregister_TPM_Callback()` and verify the thread observes it promptly (use a memory barrier or atomic).

7.3.6 Crash or segfault on first I/O call

Possible causes:

- One or more function pointers in the ops table are `NULL`.
- `Open_Channel()` was not called before I/O. Channels must be opened before `Write_To_Transport` or `Register_TPM_Callback` can be used.

Things to check:

1. Verify that your initialization function populates every entry in the function pointer table.
2. Zero-initialize your TPM struct (`memset`) before calling your init function, then verify no ops entry is `NULL` afterward. A missing assignment becomes obvious when the struct starts zeroed.
3. Step through the crash in a debugger — if the program counter is at address `0x0`, a function pointer was not set.
4. Log the return code from `Open_Channel()`. If it failed, subsequent I/O calls operate on an invalid channel.

7.3.7 GUID mismatch: connection not associated with TPM

Possible causes:

- The GUID passed to `FACE_TSS_TPM_TPMTS_Injectable_Set_Reference()` does not match the `<tpm guid='...'/>` in `TSS.xml`. These must be identical.

Things to check:

1. Search your `TSS.xml` for all `guid` attributes and confirm each one has a corresponding `Set_Reference` call with the same numeric value.
2. Log the GUID value at registration time. Compare it against your XML (remember the XML value is a string — ensure you are parsing it as the correct integer type).

3. If using multiple TPMs, verify each TPM instance uses a unique GUID and maps to the correct <connection> element.

7.3.8 SafetyBase build fails or aborts at runtime

Possible causes:

- Runtime heap allocation (`malloc`/`free`) detected. All buffers and resources must be pre-allocated during `Initialize()` or `Open_Channel()`.
- Sequence or string exceeds its bounded maximum.

Things to check:

1. Audit your TPM source for any `malloc`, `calloc`, `realloc`, `free`, `new`, or `delete` calls outside of `Initialize()` and `Open_Channel()`.
2. Verify that all sequence and string types have explicit bounds in your IDL and that your buffers are sized to those bounds.
3. Run with a heap-intercepting tool to confirm no allocations occur after initialization completes.

7.3.9 Application receives TIMED_OUT unexpectedly

Possible causes:

- Your `Read_From_Transport` or `Write_To_Transport` does not honor the `timeout` parameter. If the caller passes a finite timeout, your implementation must return `TIMED_OUT` when that duration elapses without completing the operation — not block indefinitely.
- A zero timeout means it is non-blocking. If no data is immediately available, it will return `TIMED_OUT` rather than waiting.

Things to check:

1. Log the timeout value received by your TPM function. Confirm it matches what the application passed to `Receive_Message` or `Send_Message`.
2. Verify your timeout arithmetic uses the correct units. The FACE API uses nanoseconds (`FACE_SYSTEM_TIME_TYPE`); converting incorrectly causes premature or delayed timeouts.
3. Test with `FACE_INF_TIME_VALUE` (infinite) and zero separately to confirm both blocking and non-blocking paths work correctly.

7.3.10 Application receives INVALID_PARAM on seemingly valid calls

Possible causes:

- Your TPM is returning `INVALID_PARAM` for a channel ID it does not recognize.
- A `NULL` buffer address or zero buffer capacity was passed in. Your TPM should validate these at entry and return `INVALID_PARAM`.
- Configuration name passed to `Initialize()` does not match any entry in your TPM's expected configuration.

Things to check:

1. Log all parameters at entry to the function returning `INVALID_PARAM`. Identify which validation check is triggering.
2. Print the channel ID from `Open_Channel()` and the channel ID passed to the failing call. Confirm they are the same value and type.

3. If the error comes from `Initialize()`, print the configuration name string and compare character-by-character against your XML (watch for trailing whitespace or null characters).
4. Confirm the caller is not passing an uninitialized `FACE_TSS_MESSAGE_TYPE`.

7.3.11 Register_Callback returns NO_ACTION

Possble cause:

- A callback is already registered on that channel. Only one active callback per channel is permitted.

Things to check:

1. Log inside `Register_TPM_Callback()` whether the channel's callback slot is already occupied.
2. Trace your startup sequence to determine if `Register_Callback` is being called twice for the same channel (e.g., from both the application and an initialization helper).
3. Call `Unregister_TPM_Callback()` on the existing registration before registering a new one.

7.3.12 Write or Read returns NOT_AVAILABLE

Possible causes:

- The TPM has not been initialized.
- The channel is not open. `Open_Channel()` must return `NO_ERROR` before I/O is attempted on that channel ID.
- The TPM is in a shutdown or error state. Check that `Request_TPM_State_Change()` has not transitioned the TPM to an unavailable state.

Things to check:

1. Log the TPM's internal state (uninitialized, initialized, channel-open, shutting-down) at the point where `NOT_AVAILABLE` is returned.
2. Ensure `Initialize()` completed successfully (returned `NO_ERROR`) before calling any I/O function.
3. Verify the call order: `Initialize()` → `Open_Channel()` → I/O. If any earlier step failed silently, subsequent calls will see an invalid state.
4. Check whether another thread called `Request_TPM_State_Change()` or `Close_Channel()` concurrently.
5. Check that `Request_TPM_State_Change()` has not transitioned the TPM to an unavailable state.

7.3.13 TPM returns NO_ERROR but application still reports an error

Possible causes:

- Your TPM returns `NO_ERROR` from `Write_To_Transport` even when the underlying transport failed to send.
- On receive, returning `NO_ERROR` with an incorrect `buffer_capacity` (e.g., zero) causes the TS layer to treat the message as empty.

Things to check:

1. Propagate transport-layer failures as the appropriate FACE return code so the application can detect and handle them.
2. Add error checking after every transport-layer call inside your TPM (socket send, shared-memory write, etc.) and map failures to FACE return codes.
3. Log `buffer_capacity` on the receive path immediately before returning. Confirm it equals the deserialized payload size, not the raw wire size or the buffer's total capacity.

4. Check whether the TS layer above is reporting a different error code. The mismatch between your `NO_ERROR` and the application's error indicates the TS or TA layer detected an inconsistency (e.g., size mismatch, null buffer).

7.4 Validation

To validate your third-party TPM implementation, do the following:

1. Start from a known-working *Connext TSS* example configuration and replace one element at a time.
2. Verify `Initialize -> Open_Channel -> Register_Callback -> I/O -> Unregister_Callback -> Close_Channel` in that order.
3. Run both nominal and fault-injection tests (e.g., invalid GUID, unavailable transport endpoint, malformed configuration, and/or callback timeout).
4. Test your TPM with the target FACE profile (GeneralPurpose or SafetyBase) and the target transport or middleware runtime.

8 Troubleshooting

8.1 Community Forum

RTI has created a Connext TSS Special Interest Group (SIG) on our community site for advice, suggestions, or support. Note that anything you post to this forum is open to the public; please take precaution about any information you share.

8.2 Connext Documentation

User documentation for RTI products, including *Connext Professional* 7.3.1, are available on the RTI Community Portal.

8.3 Professional Training and Services

New to DDS or *Connext*? Contact our RTI Services team or your local RTI representative to learn from our experts.

8.4 Technical Support

Phone: (408) 990-7444 Email: support@rti.com Website: https://support.rti.com/

9 API Reference

Note: The complete RTI Connext TSS C API Documentation and RTI Connext TSS C++ API Documentation are included separately in your *Connext TSS* bundle. However, this API Reference for *Connext TSS* contains examples and is easier to use for a beginner.

Note: This API section refers only to C++. For the C API, replace all instances of `::` with `_` and add the FACE object as the first argument.

9.1 Thread Safety Model

This section summarizes thread-safety behavior validated against the current library implementations for *Connext Micro* in C and C++ and *Connext Professional* in C.

9.1.1 API Thread Safety

Connext TSS is thread-safe for concurrent calls on the same Base/TypedTS instance, such as `Initialize()`, `Create_Connection()`, `Destroy_Connection()`, `Send_Message()`, `Receive_Message()`, `Register_Callback()`, and `Unregister_Callback()`.

`Set_Reference()` APIs are internally synchronized, but should still be called during initialization before message traffic starts.

`Read_Callback::Callback_Handler()` executes on middleware receive threads. Your application callback code *must* be thread-safe.

Concurrent calls are always safe, but some briefly delay others; see *Product Thread Safety* for the timing consequence.

9.1.2 Product Thread Safety

Connext TSS serializes work at two levels: at the level of *structural* operations, which consist of creating and destroying a connection, registering and unregistering a callback, initializing and finalizing the Transport Protocol Module, and constructing and destroying a Base; and at the level of *message* operations, which consist of `Send_Message()`, `Receive_Message()`, and `Is_Data_Available()`. Structural operations run one at a time, and alone. Message operations run concurrently with each other.

A message operation that is issued while a structural operation is pending, waits then completes; it never fails in this case. A healthy connection does not report an error because another thread is creating or destroying an unrelated one. The error codes these calls return always describe the connection or the caller, never the timing of an unrelated operation.

The message operation's wait is bounded by one *receive wait slice* plus the duration of the structural operation itself. A wait slice is the longest interval *Connext TSS* leaves a "receive" parked in the transport: rather than hand a blocking `Receive_Message()` its caller's timeout whole, *Connext TSS* composes that timeout from slices of `FACE_RECEIVE_WAIT_SLICE_NS` (see *DDS Defaults*) and rechecks state between them. Such a receive therefore delays a structural operation by at most one slice rather than by its full timeout, and the timeout the caller observes is unchanged. No framework lock is held across transport I/O.

Two calling contexts are refused rather than made to wait, because each would otherwise wait on the operation it is already running inside. *Connext TSS* invokes an injected Configuration implementation during a structural operation: a call back into *Connext TSS* from inside that implementation is refused, whether it is a structural or a message operation. A structural operation called from inside a callback is refused for the same reason; a message operation from a callback is not. Both refusals return `FACE::NOT_AVAILABLE`.

This serialization is process-wide per language binding: two Base instances in one process contend with each other, and the C and C++ bindings are independent. Under the SafetyBase profile, the control plane is create-only—structural operations occur during startup and not afterward—so steady-state message traffic is never paused.

9.2 Typed TS Interfaces

Connext TSS implements the FACE TSS Type-Specific interfaces in the C++ programming language.

The `RTI::TSS::Base` class also implements the FACE Configuration Injectable Interface for *Connext TSS* configuration.

For applications requiring all resources to be allocated and configured during an initialization phase, an extended function of `RTI::TSS::Base`, `Enable_Runtime()`, allows *Connext TSS* users first to create disabled DDS *Entities* during initialization and then to enable them after initialization.

- *Send_Message (Typed TS)*
- *Receive_Message (Typed TS) Function*
- *Register_Callback (Typed TS)*
- *Callback_Handler (Typed TS) Function*
- *Unregister_Callback (Typed TS) Function*

9.2.1 Headers

Include these header files to allow your application to call the FACE TSS APIs implemented by the RTI FACE TSS library.

- Given a user datatype, once the RTI FACE TSS library has been built and the *rtiddsgen* tool has been run, the generated header file `<modules>/<user_datatype_name>/TypedTS.hpp` defines the FACE TSS Type-Specific Typed Interface for the user datatype, and the generated header file `<modules>/<user_datatype_name>/TypedTS_Impl.hpp` defines the implementation of that interface.

For a reference on including headers, see the example application in the `<RTITSSHOME>\examples` directory.

9.2.2 Communication

For each user-defined type, *Connext TSS rtiddsgen* generates a class implementing the Typed-Specific Typed Interface. Its functions are responsible for sending and receiving the typed messages.

Send_Message()

```
void Send_Message(
  FACE::TSS::CONNECTION_ID_TYPE connection_id,
  FACE::TIMEOUT_TYPE timeout,
  FACE::TSS::TRANSACTION_ID_TYPE& transaction_id,
  DATATYPE_TYPE& message,
  FACE::RETURN_CODE_TYPE::Value& return_code
)
```

Table 9.1: FACE::TS::Send_Message Parameters

Parameter Name	Type	Direction	Description
connection_id	FACE::TSS::CONNECTION_ID_TYPE	IN	The ID of a created connection to use to send the message.
timeout	FACE::TIMEOUT_TYPE	IN	The `timeout` parameter is not used by *Connext TSS*.
transaction_id	FACE::TRANSACTION_ID_TYPE	INOUT	The `transaction_id` parameter is used in the request/reply communication pattern. It is not used or set by *Connext TSS*.
message	DATATYPE_TYPE	INOUT	The typed message to be sent. **NOTE**: The type of the message must correspond to the type specified in the connection configuration for the given `connection_id`.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `FACE::TS::Send_Message`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.2: FACE::TS::Send_Message Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	*Connext TSS* was not properly initialized prior to the call to `FACE::TS::Send_Message`.
FACE::INVALID_PARAM	The provided `connection_id` parameter does not correspond to an active connection.
FACE::CONNECTION_CLOSED	The provided `connection_id` existed at some point but its associated connection was already destroyed.
FACE::INVALID_MODE	The connection associated with the `connection_id` is of direction `FACE::DESTINATION`. `FACE::TS::Send_Message` only works with connections using direction `FACE::SOURCE` or `FACE::BI_DIRECTIONAL`.
FACE::NO_ACTION	The underlying TPMs returned an error.
FACE::NO_ERROR	The operation completed successfully.

Upon a successful call, the DDS *DataWriter* corresponding to the connection with the matching connection ID will have written one message.

This function is thread-safe[2].

Receive_Message()

```
void Receive_Message(
  FACE::TSS::CONNECTION_ID_TYPE connection_id,
  FACE::TIMEOUT_TYPE timeout,
  FACE::TSS::TRANSACTION_ID_TYPE& transaction_id,
  DATATYPE_TYPE& message,
  FACE::TSS::HEADER_TYPE& header,
  FACE::TSS::QoS_EVENT_TYPE& qos_parameters,
  FACE::RETURN_CODE_TYPE::Value& return_code
)
```

[2] `Send_Message()` and `Receive_Message()` are thread-safe for concurrent use. However, calls on different connection IDs have reduced contention.

Table 9.3: FACE::TS::Receive_Message Parameters

Parameter Name	Type	Direction	Description
connection_id	FACE::TSS::CONNECTION_ID_TYPE	IN	The ID of the connection to be used for receiving the message.
timeout	FACE::TIMEOUT_TYPE	IN	The duration to wait for an incoming message before returning to the calling application with a return code indicating the call has timed out.
transaction_id	FACE::TRANSACTION_ID_TYPE	INOUT	The `transaction_id` parameter is used in the request/reply communication pattern, which is not currently available in *Connext TSS*.
message	DATATYPE_TYPE	INOUT	The strongly typed message to be received. An implementer will substitute their type for `DATATYPE_TYPE`. **NOTE**: The type of the message must correspond to the type specified in the connection configuration for the given `connection_id`.
header	FACE::TSS::HEADER_TYPE	OUT	The `header` parameter is not currently set or supported by *Connext TSS*.
qos_parameters	FACE::TSS::QoS_EVENT_TYPE	OUT	The `qos_parameters` parameter is not currently set or supported by *Connext TSS*.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Receive_Message`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.4: FACE::TS::Receive_Message Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	*Connext TSS* was not properly initialized prior to the call to `FACE::TS::Receive_Message`, or no data was received, or another thread is already blocked in a receive on the same channel. Only one blocking receive per channel is admitted at a time; a second concurrent call returns immediately with this code rather than waiting alongside the first.
FACE::INVALID_PARAM	The provided `connection_id` parameter does not correspond to an active connection or the timeout value is invalid.
FACE::CONNECTION_CLOSED	The provided `connection_id` existed at some point but its associated connection was already destroyed.
FACE::INVALID_MODE	The connection associated with the `connection_id` is of direction `FACE::SOURCE`. `FACE::TS::Receive_Message` only works with connections using direction `FACE::DESTINATION` or `FACE::BI_DIRECTIONAL`.
FACE::TIMED_OUT	The call to `FACE::TS::Receive_Message` timed out before any data was received on the connection. On a connection spanning more than one channel this reports the **call's** timeout being exceeded, not any one channel's: it is returned when the timeout expires and no channel reported anything more specific. Where a channel did report something more specific – `FACE::MESSAGE_STALE` or `FACE::DATA_BUFFER_TOO_SMALL`, for instance – that verdict is returned instead, taking the first such answer in channel order so that repeated calls under identical conditions give the same result.
FACE::MESSAGE_STALE	The instance was disposed.
FACE::NO_ACTION	The underlying TPMs returned an error.
FACE::NO_ERROR	The operation completed successfully.

Users will poll with this function for received data.

This function is thread-safePage 72, 2.

9.2.3 Callback Interface

The FACE TSS callback interface allows for messages to be received without polling.

Register_Callback()

The TypedTS instance supports the Register_Callback function to receive data without polling. Each received message will invoke the registered callback.

```
void Register_Callback(
  FACE::TSS::CONNECTION_ID_TYPE   connection_id,
  Read_Callback&   callback,
  FACE::RETURN_CODE_TYPE::Value&  return_code
)
```

Table 9.5: FACE::TypedTS::Register_Callback Parameters

Parameter Name	Type	Direction	Description
connection_id	FACE::TSS::CONNECTION_ID_TYPE	IN	The ID of the connection on which the messages shall be received. This ID is returned via a successful call to `Create_Connection`.
callback	DATATYPE_TYPE::Read_Callback	IN	The user-defined callback function to invoke whenever data is received on the connection.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `FACE::TS::Register_Callback`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.6: FACE::TS::Register_Callback Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	*Connext TSS* was not properly initialized prior to the call to `Register_Callback`.
FACE::INVALID_PARAM	The provided `connection_id` parameter does not correspond to an active connection or the `callback` parameter is invalid.
FACE::CONNECTION_CLOSED	The provided `connection_id` existed at some point but its associated connection was already destroyed.
FACE::NO_ACTION	There is a callback already registered.
FACE::NO_ERROR	The callback was successfully registered.

Only one callback can be registered per connection.

A connection may be registered and unregistered repeatedly. `NO_ACTION` means a callback is registered on the connection *right now*, not that one was ever registered on it: once `Unregister_Callback` has returned successfully, a further `Register_Callback` on the same connection installs a new callback and returns `NO_ERROR`.

This function is thread-safe for concurrent callers.

Read_Callback::Callback_Handler

This function is a user-defined handler for receiving data via callback instead of polling.

```
void Callback_Handler(
  FACE::TSS::CONNECTION_ID_TYPE connection_id,
  FACE::TSS::TRANSACTION_ID_TYPE transaction_id,
  const DATATYPE_TYPE& message,
  const FACE::TSS::HEADER_TYPE& header,
  const FACE::TSS::QoS_EVENT_TYPE& qos_parameters,
  FACE::RETURN_CODE_TYPE::Value& return_code
)
```

Table 9.7: FACE::Read_Callback::Callback_Handler Parameters

Parameter Name	Type	Direction	Description
connection_id	FACE::TSS::CONNECTION_ID_TYPE	IN	The ID of the connection on which the message was received.
transaction_id	FACE::TRANSACTION_ID_TYPE	IN	The `transaction_id` parameter is used in the request/reply communication pattern, which is not currently available in *Connext TSS*.
message	FACE_SPECIFIED_TYPE	INOUT	The strongly typed message that was received. An implementer will substitute their type for `DATATYPE_TYPE`. **NOTE**: The type of the message must correspond to the type specified in the connection configuration for the given `connection_id`.
header	FACE::TSS::HEADER_TYPE	OUT	The `header` parameter is not currently set or supported by *Connext TSS*.
qos_parameters	FACE::TSS::QoS_EVENT_TYPE	OUT	The `qos_parameters` parameter is not currently set or supported by *Connext TSS*.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `FACE::TS::Receive_Message`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Unregister_Callback()

This function removes a registered callback from a connection.

```
void Unregister_Callback(
  const FACE::TSS::CONNECTION_ID_TYPE connection_id,
  FACE::RETURN_CODE_TYPE::Value   &return_code
)
```

Table 9.8: FACE::TypedTS::Unregister_Callback Parameters

Parameter Name	Type	Direction	Description
connection_id	FACE::CONNECTION_ID_TYPE	IN	The ID of the connection with the callback that will be unregistered.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Unregister_Callback`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.9: FACE::TSS::BASE::Unregister_Callback Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	The RTI TSS was not properly initialized prior to the call to `Unregister_Callback`.
FACE::INVALID_PARAM	The call failed to unregister a callback on a connection with the provided ID. This could be the result of an invalid ID or that no callback was associated with the connection.
FACE::NO_ERROR	The operation completed successfully. On a connection spanning more than one channel, this is returned if **any** channel's callback was removed; a channel whose callback the transport reports as already absent counts as removed, since the requested end state – no callback on that channel – holds either way. Channels whose callbacks could not be removed stay registered and keep dispatching, and a repeated `Unregister_Callback` retries exactly those.

This function is thread-safe for concurrent callers.

`Unregister_Callback` may return while a final callback invocation is still completing on another thread. No *new* invocation begins once it has returned: the callback is closed to further dispatch before the listener is detached, so every later delivery is discarded. What is not guaranteed is that a callback already in progress has finished. The FACE 3.2 `Unregister_Callback` semantics permit this, and *Connext TSS* takes it deliberately, in exchange for an `Unregister_Callback` that never blocks waiting on application callback code.

An application that releases resources its callback captures – a buffer the callback writes into, an object whose method it calls – must therefore tolerate one last invocation

overlapping the return of `Unregister_Callback`, and must not free those resources on the assumption that the callback is already finished. Where an application needs that assurance, it should have the callback itself signal completion, and wait for that signal after `Unregister_Callback` returns.

Calling `Unregister_Callback` – or any other control-plane operation, such as `Create_Connection`, `Destroy_Connection` or `Register_Callback` – **from inside a callback** is a programming error, and *Connext TSS* reports it rather than performing it: the call returns `FACE::NOT_AVAILABLE` and changes nothing. A callback therefore cannot unregister itself, and must ask an application thread to do it instead.

The restriction is not arbitrary. A control operation that closed the callback's own channel would delete the DDS entity whose listener is running that very callback, and deleting a DDS entity waits for a running listener to return – so the operation would wait for the callback that is waiting for the operation. Refusing is what turns that deadlock into a return code. Data-plane calls (`Send_Message`, `Receive_Message`, `Is_Data_Available`) are **not** restricted this way and may be made freely from a callback.

9.3 TSS TA Base

Connext TSS implements the FACE TSS Type-Specific Base and Typed Interfaces in the C++ programming language as the `RTI::TSS::Base` class.

`RTI::TSS::Base` also implements the FACE Configuration Injectable, TPM_Injectable, and the Logging_Injectable interfaces for *Connext TSS* configuration.

For applications requiring all resources to be allocated and configured during an initialization phase, an extended function of RTI::TSS::Base, `Enable_Runtime()`, allows *Connext TSS* users first to create disabled DDS *Entities* during initialization and then to enable them after initialization.

- *Set_Reference (Configuration)*
- *Set_Reference (TPM)*
- *Set_Reference (Logging)*
- *Initialize (Base TS) Function*
- *Create_Connection (Base TS) Function*
- *Destroy_Connection (Base TS) Function*
- *Enable_Runtime (RTI)*
- *Send_Message (Base TS)*
- *Receive_Message (Base TS) Function*
- *Register_Callback (Base TS)*
- *Callback_Handler (Base TS) Function*
- *Unregister_Callback (Base TS) Function*

9.3.1 Headers

Include these header files to allow your application to call the FACE TSS APIs implemented by the RTI FACE TSS library.

- `code/C++/include/FACE/TSS/RTI/Base_impl.hpp` defines the *Connext TSS* implementation of the FACE TSS Type-Specific Base interface.

For a reference on including headers, see the example application in the `<RTITSSHOME>\examples` directory.

9.3.2 Initialization

Set_Reference() Configuration

A new Base instance must set its reference to a Configuration Interface before the Base is initialized.

```
void Set_Reference (
    /* in */  const FACE::STRING_TYPE &interface_name,
    /* in */  FACE::Configuration **interface_reference,
    /* out */ const FACE::GUID_TYPE id,
    /* out */ FACE::RETURN_CODE_TYPE::Value &return_code);
)
```

Table 9.10: FACE::Configuration_Injectable::Set_Reference Parameters

Parameter Name	Type	Direction	Description
interface_name	FACE::STRING_TYPE	IN	Unused by *Connext TSS*.
interface_reference	FACE::Configuration	IN	A reference to a FACE Configuration instance for *Connext TSS*.
id	FACE::GUID_TYPE	OUT	Unused by *Connext TSS*.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Set_Reference(Base)`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.11: FACE::Configuration_Injectable::Set_Reference Return Codes

Return Code	Description
FACE::NO_ERROR	The operation completed successfully.
FACE::NO_ACTION	A configuration reference was previously set successfully.

The `Set_Reference` function sets the FACE Configuration instance that a Base instance will use to configure the TSS connections it will create.

The `interface_reference` must be a configured instance of FACE::Configuration implementing the FACE 3.x Configuration Interface. For help in creating and setting the configuration data in the Configuration Interface, see *Configuration Interface*.

The `Set_Reference` function is non-blocking.

This function is thread-safe for concurrent callers.

Set_Reference() TPM

If Logging is to be enabled, a new Base instance must set its reference to a Logging Interface before the Base is initialized.

```
void Set_Reference (
    /* in */  const FACE::STRING_TYPE &interface_name,
    /* in */  FACE::TSS::TPM::TPMTS **interface_reference,
    /* out */ const FACE::GUID_TYPE id,
    /* out */ FACE::RETURN_CODE_TYPE::Value &return_code);
)
```

Table 9.12: FACE::Configuration_Injectable::Set_Reference Parameters

Parameter Name	Type	Direction	Description
interface_name	FACE::STRING_TYPE	IN	Unused by *Connext TSS*.
interface_reference	FACE::TSS::TPM::TPMTS	IN	A reference to a FACE TPM instance for *Connext TSS*.
id	FACE::GUID_TYPE	OUT	Unused by *Connext TSS*.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Set_Reference(Base)`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.13: FACE::Configuration_Injectable::Set_Reference Return Codes

Return Code	Description
FACE::NO_ERROR	The operation completed successfully.
FACE::NO_ACTION	A configuration reference was previously set successfully.

The `Set_Reference` function sets the FACE TPM instance that a Base instance will use.

The `interface_reference` must be a configured instance of `FACE::TSS::TPM::TPMTS` implementing the FACE 3.x TPM interface.

The `Set_Reference` function is non-blocking.

This function is thread-safe for concurrent callers.

Set_Reference() Logging

A new Base instance must set its reference to a Logging Interface before the Base is initialized.

```
void Set_Reference (
    /* in */  const FACE::STRING_TYPE &interface_name,
    /* in */  FACE::Logging **interface_reference,
    /* out */ const FACE::GUID_TYPE id,
    /* out */ FACE::RETURN_CODE_TYPE::Value &return_code);
)
```

Table 9.14: FACE::Configuration_Injectable::Set_Reference Parameters

Parameter Name	Type	Direction	Description
interface_name	FACE::STRING_TYPE	IN	Unused by *Connext TSS*.
interface_reference	FACE::Logging	IN	A reference to a FACE Logging instance for *Connext TSS*.
id	FACE::GUID_TYPE	OUT	Unused by *Connext TSS*.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Set_Reference(Base)`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.15: FACE::Configuration_Injectable::Set_Reference Return Codes

Return Code	Description
FACE::NO_ERROR	The operation completed successfully.
FACE::NO_ACTION	A configuration reference was previously set successfully.

The `Set_Reference` function sets the FACE Logging instance that a Base instance will use to log.

The `interface_reference` must be a configured instance of `FACE::Logging` implementing the FACE 3.x Logging interface. For help in creating and setting the configuration data in the Configuration Interface, see the *Logging* section.

The `Set_Reference` function is non-blocking.

This function is thread-safe for concurrent callers.

Initialize()

Once the Configuration Interface reference is set, the TSS is initialized by calling the Base interface `Initialize(TS)` function.

```
void Initialize (
    /* in */  const FACE::CONFIGURATION_RESOURCE &configuration,
    /* out */ FACE::RETURN_CODE_TYPE::Value &return_code
)
```

Table 9.16: FACE::Base::Initialize Parameters

Parameter Name	Type	Direction	Description
return_code	FACE::RETURN_CODE_TYPE&	OUT	The return code for the call to `Initialize(TS)`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.17: FACE::TS::Initialize Return Codes

Return Code	Description
FACE::NO_ERROR	The operation completed successfully.
FACE::NO_ACTION	*Connext TSS* was successfully initialized previously.
FACE::NOT_AVAILABLE	*Connext TSS* internal data could not be properly initialized.
FACE::INVALID_CONFIG	*Connext TSS* could not be configured with the provided configuration parameter.

The `Initialize(TS)` function will initialize *Connext TSS* and its supporting TPMs with configuration data from its set Configuration Interface. It will load the system configuration the first time it is called, idempotently.

The `Initialize(TS)` function is non-blocking.

This function is thread-safe for concurrent callers.

9.3.3 Connections

Create_Connection()

A connection is created with the `RTI::TSS::Base::Create_Connection()` function.

```
void Create_Connection(
    const FACE::TSS::CONNECTION_NAME_TYPE &connection_name,
    const FACE::TIMEOUT_TYPE timeout,
    FACE::TSS::CONNECTION_ID_TYPE &connection_id,
    FACE::TSS::MESSAGE_SIZE_TYPE &max_message_size,
    FACE::RETURN_CODE_TYPE::Value &return_code
)
```

Table 9.18: FACE::Base::Create_Connection Parameters

Parameter Name	Type	Direction	Description
connection_name	FACE::TSS::CONNECTION_NAME_TYPE	IN	The name of the connection to create. Must match the name (case-insensitive) of a connection in configuration data.
timeout	FACE::TIMEOUT_TYPE	IN	Specifies a timeout for connection creation. **NOTE**: This parameter is not used by *Connext TSS*.
connection_id	FACE::TSS::CONNECTION_ID_TYPE	OUT	A unique connection ID. This ID will be used to refer to the connection in all other TSS API calls. This field is passed by reference and is set upon return from a successful call.
max_message_size	FACE::TSS::MESSAGE_SIZE_TYPE	OUT	The maximum size of a message for this connection. This field is passed by reference and is set upon return from a successful call. **NOTE**: This parameter is not supported by RTI TSS and consequently is set to 0.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `FACE::Base::Create_Connection`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.19: FACE::Base::Create_Connection Return Codes

Return Code	Description
FACE::NO_ERROR	The operation completed successfully.
FACE::NOT_AVAILABLE	Base was not properly initialized prior to the call to `FACE::TS::Create_Connection`.
FACE::INVALID_PARAM	A configuration with a name matching the provided `connection_name` parameter was not found. **NOTE**: String matching is case-insensitive, as required by the FACE TSS specification.
FACE::INVALID_CONFIG	Base was unable to create the connection due to an invalid configuration.

The `Create_Connection` function is non-blocking.

This function is thread-safe for concurrent callers.

Destroy_Connection

```
void Destroy_Connection(
    const FACE::TSS::CONNECTION_ID_TYPE connection_id,
    FACE::RETURN_CODE_TYPE::Value &return_code
)
```

Table 9.20: FACE::Base::Destroy_Connection Parameters

Parameter Name	Type	Direction	Description
connection_id	FACE::TSS::CONNECTION_ID_TYPE	IN	The ID of the connection to be destroyed.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Destroy_Connection()`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.21: FACE::TS::Destroy_Connection Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	Base was not properly initialized prior to the call to `FACE::TS::Destroy_Connection`, or at least one of the connection's channels could not be closed. See the note below on partial closes.
FACE::NO_ACTION	The connection was already destroyed
FACE::INVALID_PARAM	The connection does not exist.
FACE::NO_ERROR	The operation completed successfully. Every channel of the connection was closed and the connection no longer exists.

A connection may span more than one channel, and `Destroy_Connection` succeeds only if **every** one of them closes. `FACE::NO_ERROR` means the connection was destroyed; it is never returned for a partial close.

If any channel refuses to close, the call returns `FACE::NOT_AVAILABLE` and the connection **survives**: its record, its channel list, and the channels that are still open all remain, while the channels that did close stay closed. The connection is marked as closing from the moment the call is made, so it refuses data operations from then on – a subsequent `Send_Message` or `Receive_Message` on it returns `FACE::CONNECTION_CLOSED` – but the connection id remains valid.

The call is therefore **retryable, and should be retried**: calling `Destroy_Connection` again with the same id attempts only the channels still open, and returns `FACE::NO_ERROR` once the last of them closes. An application that treats `FACE::NOT_AVAILABLE` here as a permanent failure and does not retry leaks the connection and its open channels for the lifetime of the process.

This function is thread-safe for concurrent callers.

Enable_Runtime()

Future functionality.

9.3.4 Communication

For each user-defined type, *Connext TSS rtiddsgen* generates a class implementing the Typed-Specific Typed Interface. Its functions are responsible for sending and receiving the typed messages.

Send_Message()

```
void Send_Message(
  FACE::TSS::CONNECTION_ID_TYPE connection_id,
  FACE::TIMEOUT_TYPE timeout,
  FACE::TSS::TRANSACTION_ID_TYPE& transaction_id,
  const FACE::TSS::MESSAGE_TYPE& message,
  FACE::TSS::MESSAGE_SIZE_TYPE& size_sent,
  FACE::RETURN_CODE_TYPE::Value& return_code
)
```

Table 9.22: FACE::TSS::TypeAbstraction::TypeAbstractionTS::Send_Message Parameters

Parameter Name	Type	Direction	Description
connection_id	FACE::TSS::CONNECTION_ID_TYPE	IN	The ID of a created connection to use to send the message.
timeout	FACE::TSS::TIMEOUT_TYPE	IN	The `timeout` parameter is not used by *Connext TSS*.
transaction_id	FACE::TSS::TRANSACTION_ID_TYPE	INOUT	The `transaction_id` parameter is used in the request/reply communication pattern. It is not used or set by *Connext TSS*.
message	FACE::TSS::MESSAGE_TYPE	IN	The typed message to be sent.
size_sent	FACE::TSS::MESSAGE_SIZE_TYPE	OUT	The size sent over the wire.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `FACE::TS::Send_Message`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.23: FACE::TS::Send_Message Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	*Connext TSS* was not properly initialized prior to the call to `FACE::TS::Send_Message`.
FACE::INVALID_PARAM	The provided `connection_id` parameter does not correspond to an active connection.
FACE::CONNECTION_CLOSED	The provided `connection_id` existed at some point but its associated connection was already destroyed.
FACE::INVALID_MODE	The connection associated with the `connection_id` is in an invalid mode.
FACE::NO_ACTION	The underlying TPMs have returned an error.
FACE::NO_ERROR	The operation completed successfully.

Upon a successful call, the DDS *DataWriter* corresponding to the connection with the matching connection ID will have written one message.

This function is thread-safe[Page 72, 2].

Receive_Message()

```
void Receive_Message(
  FACE::TSS::CONNECTION_ID_TYPE connection_id,
  FACE::TIMEOUT_TYPE timeout,
  FACE::TSS::TRANSACTION_ID_TYPE& transaction_id,
  FACE::TSS::MESSAGE_SIZE_TYPE size_limit,
  FACE::TSS::MESSAGE_TYPE& message,
  FACE::TSS::HEADER_TYPE& header,
  FACE::TSS::QoS_EVENT_TYPE& qos_parameters,
  FACE::RETURN_CODE_TYPE::Value& return_code
)
```

Table 9.24: FACE::TS::Receive_Message Parameters

Parameter Name	Type	Direction	Description
connection_id	FACE::TSS::CONNECTION_ID_TYPE	IN	The ID of the connection to be used for receiving the message. This ID is returned via a successful call to `FACE::TS::Create_Connection`.
timeout	FACE::TIMEOUT_TYPE	IN	The duration to wait for an incoming message before returning to the calling application with a return code indicating the call has timed out.
transaction_id	FACE::TRANSACTION_ID_TYPE	INOUT	The `transaction_id` parameter is used in the request/reply communication pattern, which is not currently available in *Connext TSS*.
size_limit	FACE::TSS::MESSAGE_SIZE_TYPE	IN	The maximum size of a message to be received. This parameter is not currently used by *Connext TSS*.
message	FACE::TSS::MESSAGE_TYPE	INOUT	The typed message to be sent.
header	FACE::TSS::HEADER_TYPE	OUT	The `header` parameter is not currently set or supported by *Connext TSS*.
qos_parameters	FACE::TSS::QoS_EVENT_TYPE	OUT	The `qos_parameters` parameter is not currently set or supported by *Connext TSS*.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Receive_Message`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.25: FACE::TS::Receive_Message Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	*Connext TSS* was not properly initialized prior to the call to `FACE::TS::Receive_Message`, or no data was received, or another thread is already blocked in a receive on the same channel. Only one blocking receive per channel is admitted at a time; a second concurrent call returns immediately with this code rather than waiting alongside the first.
FACE::INVALID_PARAM	The provided `connection_id` parameter does not correspond to an active connection or the timeout value is invalid.
FACE::CONNECTION_CLOSED	The provided `connection_id` existed at some point but its associated connection was already destroyed.
FACE::INVALID_MODE	The connection associated with the `connection_id` is of direction `FACE::SOURCE`. `FACE::TS::Receive_Message` only works with connections using direction `FACE::DESTINATION` or `FACE::BI_DIRECTIONAL`.
FACE::TIMED_OUT	The call to `FACE::TS::Receive_Message` timed out before any data was received on the connection. On a connection spanning more than one channel this reports the **call's** timeout being exceeded, not any one channel's: it is returned when the timeout expires and no channel reported anything more specific. Where a channel did report something more specific – `FACE::MESSAGE_STALE` or `FACE::DATA_BUFFER_TOO_SMALL`, for instance – that verdict is returned instead, taking the first such answer in channel order so that repeated calls under identical conditions give the same result.
FACE::MESSAGE_STALE	The instance was disposed.
FACE::NO_ACTION	The underlying DDS-level API returned an error.
FACE::NO_ERROR	The operation completed successfully.

Users will poll with this function for received data.

Upon a successful call, the DDS *DataReader* corresponding to the connection with the matching connection ID will have received and taken one message.

This function is thread-safe[Page 72, 2].

9.3.5 Callback Interface

The FACE TSS callback interface allows for messages to be received without polling.

Register_Callback()

The Type Abstraction instance supports the Register_Callback function to receive data without polling. Each received message will invoke the registered callback.

```
void Register_Callback(
  FACE::TSS::CONNECTION_ID_TYPE connection_id,
  FACE::TSS::TypeAbstraction::Read_Callback** data_callback,
  FACE::TSS::MESSAGE_SIZE_TYPE max_message_size,
  FACE::TSS::MESSAGE_GUID_TYPE message_guid,
  FACE::RETURN_CODE_TYPE::Value& return_code
)
```

Table 9.26: FACE::TSS::TypeAbstraction::TypedAbstractionTS::Register_Callback Parameters

Parameter Name	Type	Direction	Description
connection_id	FACE::CONNECTION_ID_TYPE	IN	The ID of the connection on which the messages shall be received. This ID is returned via a successful call to `Create_Connection`.
callback	FACE::TSS::TypeAbstraction::Read_Callback::Read_Callback	INOUT	The user-defined callback function to invoke whenever data is received on the connection.
max_message_size	FACE::TSS::MESSAGE_SIZE_TYPE	IN	The `max_message_size` parameter is not currently set or supported by *Connext TSS*.
message_guid	FACE::TSS::MESSAGE_GUID_TYPE	IN	The guid associated with the message DATA_TYPE.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `FACE::TSS::TypeAbstraction::TypedAbstractionTS::Register_Callback`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.27: FACE::TS::Register_Callback Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	*Connext TSS* was not properly initialized prior to the call to `Register_Callback`.
FACE::INVALID_PARAM	The provided `connection_id` parameter does not correspond to an active connection or the `callback` parameter is invalid.
FACE::CONNECTION_CLOSED	The provided `connection_id` existed at some point but its associated connection was already destroyed.
FACE::NO_ACTION	There is a callback already registered.
FACE::NO_ERROR	The callback was successfully registered.

The registered callback is invoked in the context of the DDS *DataReader's* receive thread.

Only one callback can be registered per connection.

A connection may be registered and unregistered repeatedly. `NO_ACTION` means a callback is registered on the connection *right now*, not that one was ever registered on it: once `Unregister_Callback` has returned successfully, a further `Register_Callback` on the same connection installs a new callback and returns `NO_ERROR`.

This function is thread-safe for concurrent callers.

Read_Callback::Callback_Handler

This function is a user-defined handler for receiving data via callback instead of polling. Typed TS callbacks are triggered by a call from a Type Abstraction Callback Handler for a specific Type Abstraction channel that has been registered with the Type Abstraction. The Typed TS callback is implemented by the end user; triggering it calls the provided end user code. The Typed TS callback handlers must be registered via the callback registration FACE API on the Type Abstraction. The outline of a Typed TS callback is provided by the generated code.

```
void Callback_Handler(
  FACE::TSS::CONNECTION_ID_TYPE connection_id,
  FACE::TSS::TRANSACTION_ID_TYPE transaction_id,
  const MESSAGE_TYPE& message,
  const FACE::TSS::HEADER_TYPE& header,
  const FACE::TSS::QoS_EVENT_TYPE& qos_parameters,
  FACE::RETURN_CODE_TYPE::Value& return_code
)
```

Table 9.28: FACE::Read_Callback::Callback_Handler Parameters

Parameter Name	Type	Direction	Description
connection_id	FACE::CONNECTION_ID_TYPE	IN	The ID of the connection on which the message was received.
transaction_id	FACE::TRANSACTION_ID_TYPE	IN	The `transaction_id` parameter is used in the request/reply communication pattern, which is not currently available in *Connext TSS*.
message	MESSAGE_TYPE	INOUT	The message that was received.
header	FACE::TSS::HEADER_TYPE	OUT	The `header` parameter is not currently set or supported by *Connext TSS*.
qos_parameters	FACE::TSS::QoS_EVENT_TYPE	OUT	The `qos_parameters` parameter is not currently set or supported by *Connext TSS*.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `FACE::TS::Receive_Message`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

FACE::TSS::BASE::Unregister_Callback()

This function removes a registered callback from a connection.

```
void Unregister_Callback(
  const FACE::TSS::CONNECTION_ID_TYPE connection_id,
  FACE::RETURN_CODE_TYPE::Value   &return_code
)
```

Table 9.29: FACE::TS::Unregister_Callback Parameters

Parameter Name	Type	Direction	Description
connection_id	FACE::CONNECTION_ID_TYPE	IN	The ID of the connection with the callback that will be unregistered.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Unregister_Callback`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.30: FACE::TSS::BASE::Unregister_Callback Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	The RTI TSS was not properly initialized prior to the call to `Unregister_Callback`.
FACE::INVALID_PARAM	The call failed to unregister a callback on a connection with the provided ID. This could be the result of an invalid ID or that no callback was associated with the connection.
FACE::NO_ERROR	The operation completed successfully. On a connection spanning more than one channel, this is returned if **any** channel's callback was removed; a channel whose callback the transport reports as already absent counts as removed, since the requested end state – no callback on that channel – holds either way. Channels whose callbacks could not be removed stay registered and keep dispatching, and a repeated `Unregister_Callback` retries exactly those.

This function is thread-safe for concurrent callers.

`Unregister_Callback` may return while a final callback invocation is still completing on another thread. No *new* invocation begins once it has returned: the callback is closed to further dispatch before the listener is detached, so every later delivery is discarded. What is not guaranteed is that a callback already in progress has finished. The FACE 3.2 `Unregister_Callback` semantics permit this, and *Connext TSS* takes it deliberately, in exchange for an `Unregister_Callback` that never blocks waiting on application callback code.

An application that releases resources its callback captures – a buffer the callback writes into, an object whose method it calls – must therefore tolerate one last invocation overlapping the return of `Unregister_Callback`, and must not free those resources on the assumption that the callback is already finished. Where an application needs that assurance, it should have the callback itself signal completion, and wait for that signal after `Unregister_Callback` returns.

Calling `Unregister_Callback` – or any other control-plane operation, such as `Create_Connection`, `Destroy_Connection` or `Register_Callback` – **from inside a callback** is a programming error, and *Connext TSS* reports it rather than performing it: the call returns `FACE::NOT_AVAILABLE` and changes nothing. A callback therefore cannot unregister itself, and must ask an application thread to do it instead.

The restriction is not arbitrary. A control operation that closed the callback's own channel would delete the DDS entity whose listener is running that very callback, and deleting a DDS entity waits for a running listener to return – so the operation would wait for the callback that is waiting for the operation. Refusing is what turns that deadlock into a return code. Data-plane calls (`Send_Message`, `Receive_Message`, `Is_Data_Available`) are **not** restricted this way and may be made freely from a callback.

9.4 TSS TPM Interfaces

Note: These interfaces are internal to *Connext TSS* and are intended for Transport Protocol Module integrators. The application-facing FACE surface reaches them through the Base/Type Abstraction layer, which provides the serialization described in *Product Thread Safety*. The Transport Protocol Module layer carries no synchronization of its own: a direct caller bypasses that protection and is responsible for serializing its own calls.

Connext TSS implements the FACE TSS TPM interfaces in the C++ programming language as the `FACE::TSS::TPM::TPMTS_Impl` class.

`FACE::TSS::TPM::TPMTS_Impl` also implements the FACE Configuration Injectable Interface for *Connext TSS* configuration.

- *Set_Reference (Configuration)*
- *Set_Reference (Serialization)*
- *Set_Reference (Primitive Marshalling)*
- *Set_Reference (Logging)*
- *Initialize (TPM) Function*
- *Open_Channel (TPM) Function*
- *Close_Channel (TPM)*
- *Request_TPM_State_Change (TPM) Function*
- *Is_Data_Available (TPM)*
- *Get_TPM_Status (TPM) Function*
- *Write_To_Transport (TPM) Function*
- *Read_From_Transport (TPM) Function*
- *Register_Callback (TPM)*
- *Data_Callback_Handler (TPM) Function*
- *Event_Callback_Handler (TPM) Function*
- *Unregister_Callback (TPM) Function*

9.4.1 Headers

Include these header files to allow your application to call the FACE TSS APIs implemented by the RTI FACE TSS library.

- `code/C++/include/RTI/TSS/RTI_TPM_DDS_Micro.hpp` defines the *Connext TSS* implementation of the FACE TSS TPM interface.

For a reference on including headers, see the example application in the `<RTITSSHOME>\examples` directory.

9.4.2 Initialization

Set_Reference(Configuration)

A new TPM instance must set its reference to a Configuration Interface before the TPM is initialized.

```
void Set_Reference (
    /* in */  const FACE::STRING_TYPE &interface_name,
    /* in */  FACE::Configuration **interface_reference,
    /* out */ const FACE::GUID_TYPE id,
    /* out */ FACE::RETURN_CODE_TYPE::Value &return_code);
)
```

Table 9.31: FACE::Configuration_Injectable::Set_Reference Parameters

Parameter Name	Type	Direction	Description
interface_name	FACE::STRING_TYPE	IN	Unused by *Connext TSS*.
interface_reference	FACE::Configuration	IN	A reference to a FACE Configuration instance for *Connext TSS*.
id	FACE::GUID_TYPE	OUT	Unused by *Connext TSS*.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Set_Reference(TPM)`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.32: FACE::Configuration_Injectable::Set_Reference Return Codes

Return Code	Description
FACE::NO_ERROR	The operation completed successfully.
FACE::NO_ACTION	A configuration reference was previously set successfully.

The `Set_Reference` function sets the FACE Configuration instance that a TPM instance will use to configure the TSS channels it will create, as well as the underlying DDS.

The `interface_reference` must be a configured instance of `RTI::Configuration` implementing the FACE 3.x Configuration Interface. For help in creating and setting the configuration data in the Configuration Interface, see *Configuration Interface*.

The `Set_Reference` function is non-blocking.

This function is *not* thread-safe[1].

Set_Reference(Serialization)

If Serialization is required, a new TPM instance must set its reference to a Serialization Interface before the TPM is initialized.

```
void Set_Reference (
    /* in */  const FACE::STRING_TYPE &interface_name,
    /* in */  FACE::TSS::Serialization **interface_reference,
    /* out */ const FACE::GUID_TYPE id,
    /* out */ FACE::RETURN_CODE_TYPE::Value &return_code);
)
```

Table 9.33: FACE::TSS::Serialization_Injectable::Set_Reference Parameters

Parameter Name	Type	Direction	Description
interface_name	FACE::STRING_TYPE	IN	Unused by *Connext TSS*.
interface_reference	FACE::TSS::Serialization	IN	A reference to a FACE Serialization instance for *Connext TSS*.
id	FACE::GUID_TYPE	OUT	Unused by *Connext TSS*.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Set_Reference(TPM)`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.34: FACE::Serialization_Injectable::Set_Reference Return Codes

Return Code	Description
FACE::NO_ERROR	The operation completed successfully.
FACE::NO_ACTION	A Serialization reference was previously set successfully.

The `Set_Reference` function sets the FACE Serialization instance that a TPM instance will use.

[1] The Transport Protocol Module layer carries no synchronization of its own; it relies on the Base/Type Abstraction layer to serialize structural operations. A direct caller bypasses that serialization, so it must not call these APIs concurrently with each other, or with Transport Protocol Module data operations, in the same memory space.

The `interface_reference` must be a configured instance of `FACE::TSS::Serialization` implementing the FACE 3.x Serialization Interface.

The `Set_Reference` function is non-blocking.

This function is *not* thread-safePage 96, 1.

Set_Reference(Primitive_Marshalling)

If Serialization that requires Primitive_Marshalling is required, a new TPM instance must set its reference to a Primitive_Marshalling Interface before the TPM is initialized.

```
void Set_Reference (
    /* in */  const FACE::STRING_TYPE &interface_name,
    /* in */  FACE::TSS::Primitive_Marshalling **interface_reference,
    /* out */ const FACE::GUID_TYPE id,
    /* out */ FACE::RETURN_CODE_TYPE::Value &return_code);
)
```

Table 9.35: FACE::Primitive_Marshalling_Injectable::Set_Reference Parameters

Parameter Name	Type	Direction	Description
interface_name	FACE::STRING_TYPE	IN	Unused by *Connext TSS*.
inter-face_reference	FACE::TSS::Primitive_Marshalling	IN	A reference to a FACE Primitive_Marshalling instance for *Connext TSS*.
id	FACE::GUID_TYPE	OUT	Unused by *Connext TSS*.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Set_Reference(TPM)`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.36: FACE::Primitive_Marshalling_Injectable::Set_Reference Return Codes

Return Code	Description
FACE::NO_ERROR	The operation completed successfully.
FACE::NO_ACTION	A Primitive_Marshalling reference was previously set successfully.

The `Set_Reference` function sets the FACE Primitive_Marshalling instance that a TPM instance will use.

The `interface_reference` must be a configured instance of `FACE::TSS::Primitive_Marshalling` implementing the FACE 3.x Primitive_Marshalling Interface.

The `Set_Reference` function is non-blocking.

This function is *not* thread-safePage 96, 1.

Set_Reference(Logging_Interface)

A new TPM instance must set its reference to a Logging Interface before the TPM is initialized to enable logging.

```
void Set_Reference (
    /* in */  const FACE::STRING_TYPE &interface_name,
    /* in */  FACE::Logging **interface_reference,
    /* out */ const FACE::GUID_TYPE id,
    /* out */ FACE::RETURN_CODE_TYPE::Value &return_code);
)
```

Table 9.37: FACE::Logging::Set_Reference Parameters

Parameter Name	Type	Direction	Description
interface_name	FACE::STRING_TYPE	IN	Unused by *Connext TSS*.
inter-face_reference	FACE::Logging	IN	A reference to a FACE Logging instance for *Connext TSS*.
id	FACE::GUID_TYPE	OUT	Unused by *Connext TSS*.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Set_Reference(TPM)`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.38: FACE::Logging_Injectable::Set_Reference Return Codes

Return Code	Description
FACE::NO_ERROR	The operation completed successfully.
FACE::NO_ACTION	A Logging reference was previously set successfully.

The `Set_Reference` function sets the FACE Logging instance that a TPM instance will use.

The `interface_reference` must be a configured instance of `FACE::Logging` implementing the FACE 3.x Logging Interface.

The `Set_Reference` function is non-blocking.

This function is *not* thread-safe[Page 96, 1].

Initialize()

Once the Configuration Interface reference is set, the TPM is initialized by calling the TPM interface `Initialize(TPM)` function.

```
void Initialize (
    /* in */  const FACE::CONFIGURATION_RESOURCE &configuration,
    /* out */ FACE::RETURN_CODE_TYPE::Value &return_code
)
```

Table 9.39: FACE::TPM::Initialize Parameters

Parameter Name	Type	Direction	Description
return_code	FACE::RETURN_CODE_TYPE&	OUT	The return code for the call to `Initialize(TPM)`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.40: FACE::TS::Initialize Return Codes

Return Code	Description
FACE::NO_ERROR	The operation completed successfully.
FACE::NO_ACTION	Transport Protocol Module was successfully initialized previously.
FACE::NOT_AVAILABLE	Transport Protocol Module internal data could not be properly initialized.
FACE::INVALID_CONFIG	Transport Protocol Module could not be configured with the provided configuration parameter.

The `Initialize(TPM)` function will initialize Transport Protocol Module and its supporting DDS with configuration data from its set Configuration Interface. It will load the system configuration the first time it is called, idempotently.

The `Initialize(TPM)` function is non-blocking.

This function is *not* thread-safe[Page 96, 1].

9.4.3 Connections

Open_Channel()

A channel is created with the RTI::TSS::TPM `Open_Channel` function.

```
void Open_Channel(
  const FACE::TSS::CONNECTION_NAME_TYPE& endpoint_name,
  const FACE::TSS::DATA_BUFFER_TYPE& transport_config,
  const FACE::TSS::DATA_BUFFER_TYPE& security_config,
  FACE::TSS::TPM::CHANNEL_ID_TYPE& channel_id,
  RETURN_CODE_TYPE::Value& return_code
)
```

Table 9.41: FACE::TSS::Open_Channel Parameters

Parameter Name	Type	Direction	Description
endpoint_name	FACE::TSS::CONNECTION_NAME_TYPE	IN	The name of the channel to create. Must match the name (case-insensitive) of a connection in configuration data.
transport_config	FACE::TSS::DATA_BUFFER_TYPE	IN	Reference to implementation-defined transport metadata. **NOTE**: This parameter is not supported by RTI TSS and consequently is set to 0.
security_config	FACE::TSS::DATA_BUFFER_TYPE	OUT	Reference to implementation-defined security metadata such as certificates, public keys, or enabling encryption for a channel. **NOTE**: This parameter is not supported by RTI TSS. Security is set in Configuration.
channel_id	FACE::TSS::TPM::CHANNEL_ID_TYPE	OUT	A unique channel ID. This ID will be used to refer to the connection in all other TPM API calls for this TPM. This field is passed by reference and is set upon return from a successful call.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `FACE::TSS::TPM::Open_Channel`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.42: FACE::TSS::TPM::Open_Channel Return Codes

Return Code	Description
FACE::NO_ERROR	The operation completed successfully.
FACE::NOT_AVAILABLE	*Connext TSS* was not properly initialized prior to the call to `FACE::TSS::TPM::Open_Channel`.
FACE::INVALID_PARAM	A configuration with a name matching the provided `endpoint_name` parameter was not found. **NOTE**: Because both strings will have been converted to uppercase prior to the matching process, the matching is case-insensitive, as required by the FACE TSS specification.
FACE::INVALID_CONFIG	*Connext TSS* was unable to create the connection due to an invalid configuration.

A successfully created connection will have had DDS *Entities* asserted or created in the following manner:

Table 9.43: DDS Entity Creation

DDS Entity	Creation Requirements
DomainParticipant	Created if there is no existing *DomainParticipant* for the new connection's configured `domain` parameter. Otherwise the existing *DomainParticipant* on that domain will be reused by the new connection. See the OMG Specification for Data Distribution Service for Real-time Systems, v1.2 section 7.1.2.2.1, for information on the *DomainParticipant* entity class. **NOTE**: *DomainParticipant* reuse does not allow for new connections to modify *DomainParticipant* QoS after the initial creation of the *DomainParticipant*. Any connection created with the same domain as an existing connection will use the *DomainParticipant* QoS of the original connection, regardless of the QoS plugin specified for the new connection.
Topic	Created if there is no existing *Topic* matching the new connection's topic name in the connection configuration. Otherwise the existing *Topic* will be reused by the new connection. See the OMG Specification for Data Distribution Service for Real-time Systems, v1.2 section 7.1.2.3.2, for information on the *Topic* entity class. **NOTE**: *Topic* reuse does not allow for connections to modify *Topic* QoS after the initial creation of the *Topic*. Any connection created with the same topic name as an existing connection will use the *Topic* QoS of the original connection, regardless of the QoS plugin specified for the new connection.
Publisher	Created if the connection is configured as `FACE::SOURCE` or `FACE::BI_DIRECTIONAL`. See the OMG Specification for Data Distribution Service for Real-time Systems, v1.2 section 7.1.2.4.1, for information on the *Publisher* entity class. **NOTE**: Although DDS allows for *Publisher* to contain multiple *DataWriters*, in *Connext TSS Publisher* are not shared by connections.
Subscriber	Created if the connection is configured as `FACE::DESTINATION` or `FACE::BI_DIRECTIONAL`. See the OMG Specification for Data Distribution Service for Real-time Systems, v1.2 section 7.1.2.5.2, for information on the *Subscriber* entity class. **NOTE**: Although DDS allows for *Subscribers* to contain multiple *DataReaders*, in *Connext TSS Subscribers* are not shared by connections.
DataWriter	Created if the connection is configured as `FACE::SOURCE` or `FACE::BI_DIRECTIONAL`. See the OMG Specification for Data Distribution Service for Real-time Systems, v1.2 section 7.1.2.4.2, for information on the *DataWriter* entity class.
DataReader	Created if the connection is configured as `FACE::DESTINATION` or `FACE::BI_DIRECTIONAL`. See the OMG Specification for Data Distribution Service for Real-time Systems, v1.2 section 7.1.2.5.3, for information on the *DataReader* entity class.

The `Open_Channel` function is non-blocking.

This function is *not* thread-safe[Page 96, 1].

Close_Channel

```
void Close_Channel(
    const FACE::TSS::CONNECTION_ID_TYPE channel_id,
    FACE::RETURN_CODE_TYPE::Value &return_code
)
```

Table 9.44: FACE::TSS::TPM::Close_Channel Parameters

Parameter Name	Type	Direction	Description
channel_id	FACE::TSS::TPM::CHANNEL_ID_TYPE	IN	The ID of the connection to be destroyed.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Close_Channel()`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.45: FACE::TSS::TPM::Close_Channel Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	*Connext TSS* was not properly initialized prior to the call to `FACE::TS::Destroy_Connection`.
FACE::NO_ACTION	The connection was already destroyed.
FACE::INVALID_PARAM	The connection does not exist.
FACE::NO_ERROR	The operation completed successfully.

Note: Transport Protocol Module built with *Connext Micro* for Safety Base or Security OSS profiles does not reclaim any Transport Protocol Module channels or DDS resources.

9.4.4 TPM Status and State

Request_TPM_State_Change

```
void Request_TPM_State_Change(
  const TPM_STATE_TYPE::Value& new_state,
  const FACE::TSS::TPM::TPMTS::STATE_CHANGE_DATA_TYPE& data,
  RETURN_CODE_TYPE::Value& return_code
  )
```

Table 9.46: FACE::TSS::TPM::Request_TPM_State_Change Parameters

Parameter Name	Type	Direction	Description
new_state	FACE::TSS::TPM_STATE_TYPE	IN	The new state to which the TPM should transition.
data	FACE::TSS::TPM::TPMTS::STATE_CHANGE_DATA_TYPE	IN	Reference to data associated with requests to change the test or secure states.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Request_TPM_State_Change()`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.47: FACE::TSS::TPM::Request_TPM_State_Change

Return Code	Description
FACE::NOT_AVAILABLE	Transport Protocol Module does support the requested state change.
FACE::NO_ACTION	The connection was already destroyed.
FACE::NO_ERROR	The operation completed successfully.

Is_Data_Available

```
void Is_Data_Available(
  CHANNEL_ID_SEQ_TYPE channel_ids,
  FACE::TIMEOUT_TYPE timeout,
  CHANNEL_ID_SEQ_TYPE available_ids,
```

(continues on next page)

(continued from previous page)

```
RETURN_CODE_TYPE return_code);
)
```

Table 9.48: FACE::tss::TPM::TPMTS::Is_Data_Available

Parameter Name	Type	Direction	Description
channel_ids	FACE::tss::TPM::TPMTS::CHANNEL_ID_SEQ_TYPE	IN	The list of channels to be checked for data.
timeout	FACE::TIMEOUT_TYPE	IN	The duration to wait for an incoming message before returning to the calling application with a return code indicating the call has timed out.
available_ids	FACE::tss::TPM::TPMTS::CHANNEL_ID_SEQ_TYPE	OUT	The list of channels for which data is available.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Is_Data_Available()`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.49: FACE::TS::Is_Data_Available Return Codes

Return Code	Description
FACE::NO_ACTION	Transport Protocol Module had an unknown failure.
FACE::NOT_AVAILABLE	Transport Protocol Module was not properly initialized prior to the call to `FACE::tss::TPM::TPMTS::Is_Data_Available()`.
FACE::INVALID_PARAM	The list of channel_ids was null or did not contain any valid ids.
FACE::NO_ERROR	The operation completed successfully.

Note: Transport Protocol Module built with *Connext Micro* for Safety Base or Security OSS profiles does not reclaim any *Connext TSS* connection or DDS resources.

Get_TPM_Status

```
void Get_TPM_Status(
  EVENT_TYPE::Value& status,
  RETURN_CODE_TYPE::Value& return_code
  )
```

Table 9.50: FACE::TSS::TPM::TPMTS::Get_TPM_Status Parameters

Parameter Name	Type	Direction	Description
status	FACE::TSS::TPM::EVENT_TYPE::Value	out	The ID of the connection to be destroyed.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Get_TPM_Status()`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.51: FACE::TS::Get_TPM_Status Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	Transport Protocol Module was not properly initialized prior to the call to `FACE::TSS::TPM::TPMTS::Get_TPM_Status()`.
FACE::NO_ERROR	The operation completed successfully.

Note: Transport Protocol Module built with *Connext Micro* for Safety Base or Security OSS profiles does not reclaim any *Connext TSS* connection or DDS resources.

9.4.5 Communication

For each user-defined type, *Connext TSS rtiddsgen* generates a class implementing the Typed-Specific Typed Interface. Its functions are responsible for sending and receiving the typed messages.

Write_To_Transport()

```
void Write_To_Transport(
  FACE::TSS::TPM::CHANNEL_ID_TYPE channel_id,
  FACE::TIMEOUT_TYPE max_delay,
  const FACE::TSS::MESSAGE_TYPE& message,
  const FACE::TSS::HEADER_TYPE& TSS_header,
  FACE::TSS::TRANSACTION_ID_TYPE transaction_id,
  FACE::RETURN_CODE_TYPE::Value& return_code
  )
```

Table 9.52: FACE::TSS::TPM::TPMTS::Write_To_Transport Parameters

Parameter Name	Type	Direction	Description
channel_id	FACE::TSS::TPM::CHANNEL_ID_TYPE	IN	The ID of a created connection to use to send the message.
max_delay	FACE::TIMEOUT_TYPE	IN	The `timeout` parameter is not used by *Connext TSS*.
message	FACE::TSS::MESSAGE_TYPE	INOUT	The message to be sent.
TSS_Header	FACE::TSS::HEADER_TYPE	INOUT	The `header` parameter is not used by *Connext TSS*.
transaction_id	FACE::TRANSACTION_ID_TYPE	INOUT	The `transaction_id` parameter is used in the request/reply communication pattern. It is not used or set by *Connext TSS*.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `FACE::TSS::TPM::TPMTS::Write_To_Transport`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.53: FACE::TSS::TPM::TPMTS::Write_To_Transport Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	Transport Protocol Module was not properly initialized prior to the call to `FACE::TSS::TPM::TPMTS::Write_To_Transport`.
FACE::INVALID_PARAM	The provided `channel_id` parameter does not correspond to an active channel or the DDS *DataWriter's* write call returned an error.
FACE::CONNECTION_CLOSED	The provided `channel_id` existed at some point but its associated channel was already destroyed.
FACE::NO_ACTION	The underlying DDS-level API returned an error.
FACE::NO_ERROR	The operation completed successfully.

Upon a successful call, the DDS *DataWriter* corresponding to the channel with the matching channel ID will have written one message.

This function is thread-safe against concurrent data flow: it may run concurrently with itself and with `Read_From_Transport()`. It is *not* safe against a concurrent Transport Protocol Module control operation issued by a direct caller.

Read_From_Transport()

```
void Read_From_Transport(
  FACE::TSS::TPM::CHANNEL_ID_TYPE channel_id,
  FACE::TIMEOUT_TYPE timeout,
  FACE::TSS::TRANSACTION_ID_TYPE& transaction_id,
  FACE::TSS::MESSAGE_TYPE& message,
  FACE::TSS::HEADER_TYPE& TSS_header,
  FACE::TSS::QoS_EVENT_TYPE& qos_parameters,
  RETURN_CODE_TYPE::Value& return_code
)
```

Table 9.54: FACE::TSS::TPM::TPMTS::Read_From_Transport Parameters

Parameter Name	Type	Direction	Description
connection_id	FACE::TSS::TPM::CHANNEL_ID_TYPE	IN	The ID of the channel to be used for receiving the message. This ID is returned via a successful call to `FACE::TSS::TPM::TPMTS::Open_Channel`.
timeout	FACE::TIMEOUT_TYPE	IN	The duration to wait for an incoming message before returning to the calling application with a return code indicating the call has timed out.
transaction_id	FACE::TSS::TRANSACTION_ID_TYPE	INOUT	The `transaction_id` parameter is used in the request/reply communication pattern, which is not currently available in *Connext TSS*.
message	FACE::TSS::MESSAGE_TYPE	INOUT	The message to be received.
header	FACE::TSS::HEADER_TYPE	OUT	The `header` parameter is not currently set or supported by *Connext TSS*.
qos_parameters	FACE::TSS::QoS_EVENT_TYPE	OUT	The `qos_parameters` parameter is not currently set or supported by *Connext TSS*.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `FACE::TSS::TPM::TPMTS::Read_From_Transport`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.55: FACE::TSS::TPM::TPMTS::Read_From_Transport Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	Transport Protocol Module was not properly initialized prior to the call to `FACE::TSS::TPM::TPMTS::Read_From_Transport` or no data was received.
FACE::INVALID_PARAM	The provided `channel_id` parameter does not correspond to an active channel or the timeout value is invalid.
FACE::CONNECTION_CLOSED	The provided `channel_id` existed at some point but its associated channel was already destroyed.
FACE::INVALID_MODE	The channel associated with the `channel_id` is of direction `FACE::SOURCE`. `FACE::TSS::TPM::TPMTS::Read_From_Transport` only works with channels using direction `FACE::DESTINATION` or `FACE::BI_DIRECTIONAL`.
FACE::TIMED_OUT	The call to `FACE::TSS::TPM::TPMTS::Read_From_Transport` timed out before any data was received on the channel.
FACE::MESSAGE_STALE	The instance was disposed.
FACE::NO_ACTION	The underlying DDS-level API returned an error.
FACE::NO_ERROR	The operation completed successfully.

Users will poll with this function for received data.

Upon a successful call, the DDS *DataReader* corresponding to the channel with the matching channel ID will have received and taken one message.

This function is thread-safe against concurrent data flow: it may run concurrently with itself and with `Write_To_Transport()`. It is *not* safe against a concurrent Transport Protocol Module control operation issued by a direct caller.

9.4.6 Callback Interface

Type Abstraction callbacks are triggered by a call from a Transport Protocol Module Data Callback Handler for a specific Type Abstraction connection. The Type Abstraction callback is internally implemented in the RTI implementation of the Type Abstraction; triggering it calls the associated Typed TS Callback handler. Type Abstraction callback registration happens automatically when a Typed TS callback is registered.

Register_Callback()

The TypedTS instance supports the Register_Callback function to receive data without polling. Each received message will invoke the registered callback.

```
void Register_TPM_Callback(
  FACE::TSS::CONNECTION_ID_TYPE   channel_id,
  FACE::TSS::TPM::TPM_Callback**   callback,
  FACE::RETURN_CODE_TYPE::Value&  return_code
)
```

Table 9.56: FACE::TSS::TPM::TMPTS::Register_TPM_Callback Parameters

Parameter Name	Type	Direction	Description
channel_id	FACE::CONNECTION_ID_TYPE	IN	The ID of the channel on which the messages shall be received. This ID is returned via a successful call to `Open_Channel`.
callback	FACE::TSS::TPM::TPM_Callback	IN	The user-defined callback function(s) to invoke.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `FACE::TSS::TPM::TMPTS::Register_TPM_Callback`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.57: FACE::TSS::TPM::TMPTS::Register_TPM_Callback Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	Transport Protocol Module was not properly initialized prior to the call.
FACE::INVALID_PARAM	The provided `channel_id` parameter does not correspond to an active channel or the `callback` parameter is invalid.
FACE::CONNECTION_CLOSED	The provided `channel_id` existed at some point but its associated channel was already destroyed.
FACE::NO_ACTION	There is a callback already registered.
FACE::NO_ERROR	The callback was successfully registered.

The registered callback is invoked in the context of the DDS *DataReader's* receive thread.

Only one callback can be registered per channel.

This function is *not* thread-safe[Page 96, 1].

TPM_Callback::Data_Callback_Handler

TPM Data callbacks are called on DDS Data events which are triggered by the DDS Listener on_data_available callback. The end user is responsible for implementing the DDS Data on_data_available callback and a generated example is provided for each of the high level types. The on_data_available callback calls the TPM Data Callback handler. The TPM Data Callback handler is implemented by the end user and must be injected into the TPM via FACE Injectable APIs for it to be used. RTI TSS provides a default implementation of the TPM Data Callback handler that calls up to Type Abstraction->Typed TS layers. When setting up a Data Callback, the end user must register the on_data_available callback with the DDS Listener in the Transport Protocol Module configuration. The underlying *Connext* APIs should be consulted for more information about registering a DDS Listener on_data_available callback.

```
void Data_Callback_Handler(
  FACE::TSS::TPM::CHANNEL_ID_TYPE channel_id,
  FACE::TSS::TRANSACTION_ID_TYPE transaction_id,
  const FACE::TSS::MESSAGE_TYPE& message,
  const FACE::TSS::HEADER_TYPE& tss_header,
  const FACE::TSS::QoS_EVENT_TYPE& qos_parameters,
  RETURN_CODE_TYPE::Value& return_code
  )
```

Table 9.58: FACE::Read_Callback:Data_Callback_Handler_Parameters

Parameter Name	Type	Direction	Description
channel_id	FACE::TSS::TPM::CHANNEL_ID_TYPE	IN	The ID of the channel on which the message was received.
transaction_id	FACE::TSS::TRANSACTION_ID_TYPE	IN	The `transaction_id` parameter is used in the request/reply communication pattern, which is not currently available in *Connext TSS*.
message	FACE::TSS::MESSAGE_TYPE	INOUT	The message that was received.
header	FACE::TSS::HEADER_TYPE	OUT	The `header` parameter is not currently set or supported by *Connext TSS*.
qos_parameters	FACE::TSS::QoS_EVENT_TYPE	OUT	The `qos_parameters` parameter is not currently set or supported by *Connext TSS*.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Data_Callback_Handler`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.59: FACE::Unregister_Callback Return Codes

Return Code	Description
FACE::NO_ERROR	The callback was successfully called.
FACE::INVALID_PARAM	One of the parameters was invalid.

TPM_Callback::Event_Callback_Handler

Event callbacks are called on DDS Listener events with the end user being responsible for implementing the DDS Listener Event callbacks. In this DDS Listener callback the user calls the TPM Event Callback handler that has been injected into the TPM. The event Callback handler also implemented by the end user will be called and the user code executed upon DDS Listener event callbacks being triggered. For a list of example DDS Listener events see the DDS Listener API documentation. When setting up an Event Callback, the end user must register the DDS Listener event callbacks with the DDS Listener in the Transport Protocol Module configuration. The underlying *Connext* APIs should be consulted for more information about registering a DDS Listener event callback.

```
void Event_Callback_Handler(
  FACE::TSS::TPM::CHANNEL_ID_TYPE channel_id,
  FACE::TSS::TRANSACTION_ID_TYPE transaction_id,
  const EVENT_TYPE::Value& event,
  FACE::TSS::TPM::TPM_Callback::EVENT_CODE_TYPE event_code,
  const FACE::TSS::TPM::TPM_Callback::DIAGNOSTIC_MSG_TYPE& diagnostic_msg,
  RETURN_CODE_TYPE::Value& return_code);
```

Table 9.60: FACE::Read_Callback:Event_Callback_Handler_Parameters

Parameter Name	Type	Direction	Description
channel_id	FACE::TSS::TPM::CHANNEL_ID_TYPE	IN	The ID of the channel on which the message was received.
transaction_id	FACE::TRANSACTION_ID_TYPE	IN	The `transaction_id` parameter is used in the request/reply communication pattern, which is not currently available in *Connext TSS*.
event	EVENT_TYPE::Value	IN	The event that was received.
event_code	FACE::TSS::TPM::TPM_Callback::EVENT_CODE_TYPE	OUT	Event Code returned by the underlying transport.
diagnostic_msg	FACE::TSS::TPM::TPM_Callback::DIAGNOSTIC_MSG_TYPE	OUT	The associated diagnostic message from the event.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Event_Callback_Handler`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

FACE::TSS::BASE::Unregister_Callback()

This function removes a registered callback from a connection.

```
void Unregister_Callback(
  const FACE::TSS::CONNECTION_ID_TYPE connection_id,
  FACE::RETURN_CODE_TYPE::Value   &return_code
)
```

Table 9.61: FACE::TS:Unregister_Callback Parameters

Parameter Name	Type	Direction	Description
connection_id	FACE::CONNECTION_ID_TYPE	IN	The ID of the connection with the callback that will be unregistered.
return_code	FACE::RETURN_CODE_TYPE	OUT	The return code for the call to `Unregister_Callback`. Upon return of the call, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.62: FACE::TSS::BASE::Unregister_Callback Return Codes

Return Code	Description
FACE::NOT_AVAILABLE	The RTI TSS was not properly initialized prior to the call to `FACE::TS::Receive_Message` or no data was received.
FACE::INVALID_PARAM	The call failed to unregister a callback on a connection with the provided ID. This could be the result of an invalid ID or that no callback was associated with the connection.
FACE::INVALID_PARAM	The provided `connection_id` parameter does not correspond to an active connection or the `data_callback` parameter is invalid.

This function is *not* thread-safePage 96, 1.

9.4.7 FACE String

Connext TSS provides a FACE compliant String API.

- *Constructors*
- *Destructor*

- *clear*
- *append*
- *reserve*
- *buffer*
- *length*
- *capacity*
- *bound*
- *is_managed*
- *is_bounded*
- *is_valid*

Headers

Include `code/C++/include/FACE/String.hpp` to allow your application to call the FACE String APIs implemented by the RTI FACE String library. This file defines the *Connext TSS* implementation of the FACE String Interface.

For a reference on including headers, see the example application in the `<RTITSSHOME>\examples` directory.

FACE::String::String()

Default Constructor:

```
String()
```

Managed Constructor:

```
String(
  FACE::UnsignedLong bound,
  RETURN_CODE& return_code)
```

Unmanaged Constructor:

```
String(const char * str)
```

Managed Copy Constructor:

```
String(const String& str)
```

Managed C-String Constructor:

```
String(
 const char *str,
 RETURN_CODE& return_code)
```

Unmanaged Constructor:

```
String(
  char* str,
  FACE::UnsignedLong length,
  FACE::UnsignedLong bound,
  RETURN_CODE& return_code)
```

FACE::String::~String()

Default Destructor:

```
~String()
```

FACE::String::clear()

Clears this string's data:

```
void clear()
```

FACE::String::append()

Appends a string or an element to the string:

```
RETURN_CODE append(const String& str)
```

```
RETURN_CODE append(const FACE::Char& elem)
```

FACE::String::reserve()

Attempts to reserve memory for `capacity` number of characters:

```
RETURN_CODE reserve(FACE::UnsignedLong capacity);
```

FACE::String::buffer()

Returns C-string representation of string data:

```
char * buffer();
const char * buffer() const;
```

FACE::String::length()

Returns the length of this String:

```
FACE::UnsignedLong length() const;
```

FACE::String::capacity()

Returns the capacity of this String:

```
FACE::UnsignedLong capacity() const;
```

FACE::String::bound()

Returns the bound of this String:

```
FACE::UnsignedLong bound() const;
```

FACE::String::is_managed()

Returns whether or not this String is managed:

```
FACE::Boolean is_managed() const;
```

FACE::String::is_bounded()

Returns whether or not this String is bounded:

```
FACE::Boolean is_bounded() const;
```

FACE::String::is_valid()

Returns whether or not this String is in the invalid state:

```
FACE::Boolean is_valid() const;
```

9.4.8 FACE Sequence

Connext TSS provides a FACE compliant Sequence API.

- *Constructors*
- *Destructor*
- *clear*
- *append*

- *reserve*
- *buffer*
- *length*
- *capacity*
- *bound*
- *is_managed*
- *is_bounded*
- *is_valid*

Headers

Include `code/C++/include/FACE/Sequence.hpp` to allow your application to call the FACE Sequence APIs implemented by the RTI FACE Sequence library. This file defines the TSS implementation of the FACE Sequence Interface.

For a reference on including headers, see the example application in the `<RTITSSHOME>\examples` directory.

FACE::Sequence::Sequence()

Default Constructor:

```
Sequence()
```

Managed Constructor:

```
Sequence(
  FACE::UnsignedLong bound,
  RETURN_CODE& return_code)
```

Managed C-style array Constructor:

```
Sequence(const T * arr,
              FACE::UnsignedLong length,
              RETURN_CODE& return_code)
```

Managed Copy Constructor:

```
Sequence(const Sequence& seq)
```

Unmanaged Constructor:

```
Sequence(
  const T * arr,
  FACE::UnsignedLong length,
  FACE::UnsignedLong bound,
  RETURN_CODE& return_code)
```

FACE::Sequence::~Sequence()

Default Destructor:

```
~Sequence()
```

FACE::Sequence::clear()

Clears this Sequence's data:

```
void clear()
```

FACE::Sequence::append()

Appends a Sequence or an element to the Sequence:

```
RETURN_CODE append(const Sequence& seq)
```

```
RETURN_CODE append(const T& elem)
```

FACE::Sequence::reserve()

Attempts to reserve memory for `capacity` number of characters:

```
RETURN_CODE reserve(FACE::UnsignedLong capacity);
```

FACE::Sequence::buffer()

Returns C-Sequence representation of Sequence data:

```
T * buffer();
const T * buffer() const;
```

FACE::Sequence::length()

Returns the length of this Sequence:

```
FACE::UnsignedLong length() const;
```

FACE::Sequence::capacity()

Returns the capacity of this Sequence:

```
FACE::UnsignedLong capacity() const;
```

FACE::Sequence::bound()

Returns the bound of this Sequence:

```
FACE::UnsignedLong bound() const;
```

FACE::Sequence::is_managed()

Returns whether or not this Sequence is managed:

```
FACE::Boolean is_managed() const;
```

FACE::Sequence::is_bounded()

Returns whether or not this Sequence is bounded:

```
FACE::Boolean is_bounded() const;
```

FACE::Sequence::is_valid()

Returns whether or not this Sequence is in the invalid state:

```
FACE::Boolean is_valid() const;
```

9.5 Logging

Connext TSS utilizes the logging interface found in FACE CR 748 to implement logging. *Connext TSS* by defaults does not log until a Logging Injectable has been injected into each UoC that requires logging to be enabled. To log, *Connext TSS* calls the injected logger via the FACE logging APIs. You are responsible for providing the FACE Log type to be injected.

The *Hello_Goodbye Example*, included in *Connext TSS* in both C and C++, provides a basic FACE logger that prints any log message sent to it regardless of logging level. You can choose to modify the example or provide your own logger.

Note: If no FACE Logger is injected, *Connext TSS* will not perform any logging.

9.5.1 Log_Text(LOGGING)

```
module FACE {
interface Logging {
void Log_Text (
in TEXT_MESSAGE_TYPE message,
in SEVERITY_LEVEL severity,
out RETURN_CODE_TYPE return_code
}; // interface Logging
}; module FACE
```

Table 9.63: FACE Log_Text(LOGGING) Parameters

Parameter Name	Description
message	Contains the log message to be logged.
severity	Contains the applicable logging level of the message.
return_code	Upon return, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.64: FACE Log_Text(LOGGING) Return Codes

Return Code	Description
NO_ERROR	Indicates successful completion of the operation.
INVALID_CONFIG	Indicates that the interface is not properly configured for use.
INVALID_PARAM	Indicates that either the buffer or return_code pointer (in appropriate languages) is invalid or the message is of zero length.

9.5.2 Log_Binary(LOGGING)

```
module FACE {
interface Logging {
void Log_Binary (
in SYSTEM_ADDRESS_TYPE message,
in LENGTH_TYPE length,
in SEVERITY_LEVEL severity,
```

(continues on next page)

(continued from previous page)

```
out RETURN_CODE_TYPE return_code
}; // interface Logging
}; module FACE
```

Table 9.65: FACE Log_Binary(LOGGING) Parameters

Parameter Name	Description
buffer	Contains the binary log message to be logged.
severity	Specifies the applicable logging level of the message.
return_code	Upon return, this parameter will be populated with a value that will indicate either success or a reason for failure.

Table 9.66: FACE Log_Binary(LOGGING) Return Codes

Return Code	Description
NO_ERROR	Indicates successful completion of the operation.
INVALID_CONFIG	Indicates that the interface is not properly configured for use.
INVALID_PARAM	Indicates that either the buffer or return_code pointer (in appropriate languages) is invalid or the message is of zero length.

9.6 TSS Configuration Defaults

Connext TSS provides a central configuration header file, `RTI_Connext_TSS_Config.h`, located in the `code/` directory. This file defines default values for compile-time parameters that control resource limits, buffer sizes, hash-table tuning, and other internal constants.

Every definition is guarded with `#ifndef` so that you can override any value by passing a `-D` flag to the compiler (or via CMake `add_definitions` / `target_compile_definitions`). If no override is provided the default value is used.

Include the configuration header in your source files with the following code:

```
#include "RTI_Connext_TSS_Config.h"
```

The following subsections define the default parameters in more detail.

9.6.1 Type Bounds

These defaults control the maximum sizes of unbounded IDL strings and sequences generated by *rtiddsgen*.

Table 9.67: Type Bound Defaults

Define	Default	Description
STRINGS_BOUND	255	Maximum length of unbounded strings.
SEQUENCES_BOUND	4096	Maximum element count of unbounded sequences.

9.6.2 Resource Limits

These limits govern the maximum number of connections, channels, callbacks, and TPM references managed by *Connext TSS*.

Table 9.68: Resource Limit Defaults

Define	Default	Description
MAX_CONNECTIONS_PER_BASE_INSTANCE	10000	Maximum connections a single Base instance can own.
MAX_TPM_REFERENCES_PER_BASE_INSTANCE	50	Maximum Transport Protocol Module references a Base instance can hold.
MAX_CHANNELS_PER_CONNECTION	10000	Maximum channels within a single connection.
FACE_MAX_CHANNELS_PER_MANAGER	50	Maximum channels per channel manager (used when no system configuration specifies a different value).
MAX_CALLBACKS_PER_CONNECTION	32	Maximum callbacks that can be registered on a single connection.

9.6.3 Buffer Sizes

These defines set the sizes of internal buffers used for messages, configuration strings, and logging.

Table 9.69: Buffer Size Defaults

Define	Default	Description
FACE_DEFAULT_BUFFER_SIZE	256	General-purpose buffer for log messages, temporary strings, etc.
FACE_DEFAULT_MESSAGE_SIZE	1024	Message buffer size.
FACE_LARGE_BUFFER_SIZE	8192	Buffer for configuration strings and bulk data operations.
MAX_CONFIGURATION_STRING_LENGTH	1024	Maximum length of the TSS XML configuration string. Increase this if your configuration exceeds the default.
LOG_FILE_NAME_MAX_LENGTH	256	Maximum length of the log file name when logging is enabled.
FACE_LOG_BUFFER_SIZE	4096	Size of the internal log message buffer.

9.6.4 Hash Table Tuning

Connext TSS uses hash tables internally for entity lookup. These constants control initial sizing, growth, and the hash algorithms used.

Table 9.70: Hash Table Defaults

Define	Default	Description
FACE_DEFAULT_HASH_TABLE_SIZE	128	Initial bucket count for hash tables.
FACE_MIN_HASH_TABLE_SIZE	64	Lower bound for hash table bucket counts.
HASH_TABLE_LOAD_FACTOR_PERCENT	75	Load factor percentage that triggers rehashing.
HASH_TABLE_GROWTH_FACTOR	2	Multiplier applied when expanding hash table capacity.
HASH_MAX_ITERATIONS	100	Maximum iterations for hash operations (prevents infinite loops).
DJB2_HASH_INIT	5381	Initial value for the DJB2 hash algorithm.
DJB2_HASH_MULT	33	Multiplier for the DJB2 hash calculation (`hash * 33 + c`).
FNV_OFFSET_BASIS	0x811c9dc5	Initial value for the FNV-1a hash algorithm.
FNV_PRIME	0x01000193	Prime multiplier for the FNV-1a hash calculation.

9.6.5 DDS Defaults

This section defines the default DDS configuration values used by *Connext TSS* when no explicit values are provided.

Table 9.71: DDS Defaults

Define	Default	Description
FACE_DEFAULT_DOMAIN_ID	0	DDS domain ID used when no specific domain is configured.
FACE_DEFAULT_TIMEOUT_MS	1000	Default timeout in milliseconds for general operations.
FACE_RECEIVE_WAIT_SLICE_NS	100000000	Length in nanoseconds of one bounded wait slice. A blocking Receive_Message never hands its caller's timeout to the transport whole: *Connext TSS* composes that timeout from slices of this length and rechecks the connection's state between them. The timeout a caller observes is unchanged. What this value bounds is how long destroying a connection, or any other operation that must pause the data plane, waits for a receive that is parked in the transport. Lowering it shortens that wait at the cost of one waitset round-trip per slice per blocked receive; raising it does the reverse. Message latency is unaffected either way, because a waitset returns as soon as data arrives.

9.6.6 Overriding Defaults

To override any value at build time by defining it before the header is included (or via a compiler flag), use one of the following commands:

```
# CMake
add_definitions(-DMAX_CONNECTIONS_PER_BASE_INSTANCE=500)

# Command line
gcc -DMAX_CONNECTIONS_PER_BASE_INSTANCE=500 ...
```

Or, include the following in your source before the include:

```
#define MAX_CONNECTIONS_PER_BASE_INSTANCE 500
#include "RTI_Connext_TSS_Config.h"
```

10 Release Notes

10.1 Language Support

Connext TSS supports FACE TSS C and C++ APIs for the *Connext Micro* Transport Protocol Module[1]. *Connext TSS* supports FACE TSS C API for the *Connext Professional* Transport Protocol Module.

[1] On LynxOS-178, the C++ API for the *Connext Micro* Transport Protocol Module has not been tested by RTI.

10.2 Supported Platforms

Connext TSS has been demonstrated with *Connext Professional* 7.3.1 on the following platforms:

Table 10.1: Connext Professional - Demonstrated Architectures

OS	Version	CPU	Compiler	*Connext TSS* Architecture Abbreviation	Connext Professional Architecture Abbreviation	OSS Profile
VxWorks	VxWorks 23.09	x64	llvm 16.0	x64Vx23.09llvm16.0_rtpFACE_GP	x64Vx23.09llvm16.0_rtp	GeneralPurpose
Linux	Ubuntu 24.04 LTS	x64	gcc 13.3.0	x64Linux6gcc13.3.0FACE_GP	x64Linux4gcc7.3.0FACE_GP	GeneralPurpose
Windows[2]	Windows 11	x64	VS2022	x64Win64VS2022FACE_GP	x64Win64VS2017	GeneralPurpose

Connext TSS has also been demonstrated with *Connext Micro* 2.4.14.3 on the following platforms:

Table 10.2: Connext Micro - Demonstrated Architectures

OS	Version	CPU	Compiler	*Connext TSS* Architecture Abbreviation	Connext Micro Architecture Abbreviation	OSS Profile
Linux	Ubuntu 24.04 LTS	x64	gcc 13.3.0	x64Linux6gcc13.3.0FACE_GP	x64Linux6gcc13.3.0	GeneralPurpose
	Ubuntu 24.04 LTS	x64	gcc 13.3.0	x64Linux6gcc13.3.0FACE_SB	x64Linux4gcc7.3.0CERT	SafetyBase
LynxOS-178	LynxOS-178 2024.09.0 (APEX API)	x64	gcc 11.3.0	x64Lynx1782024.09.0.APEXgcc11.3.0FACE_SB	x64Lynx1782024.09.0.APEXgcc11.3.0	SafetyBase
Windows[Page 128, 23]	Windows 11	x64	VS2022	x64Win64VS2022FACE_GP	x64Win64VS2022	GeneralPurpose

[2] Windows is not supported by FACE. Therefore, *Connext TSS* customer support is limited. Our support and services team can diagnose issues but we do not attempt to close the gaps in the FACE standard.
[3] Debug builds of the *Connext TSS* Micro TPMs on Windows are not supported. Therefore, *Connext TSS* customer support is limited.

10.3 What's New in 4.2.0

10.3.1 Support for FACE 3.2 Technical Standard

This release upgrades *Connext TSS* to support the FACE Technical Standard, Edition 3.2.

10.3.2 Protect thread safety in TSS applications

This release adds several thread safety features to the TSS API: structural operations—creating and destroying connections, registering and unregistering callbacks—are serialized process-wide, while sends and receives run concurrently.

An upgrading application can observe two behaviors:

- A structural operation briefly delays message traffic. Data operations that are issued while one is pending, wait and then complete; they do not fail.
- `Unregister_Callback()` may return while a final callback invocation is still completing, though no new invocation begins after it returns.

For more information, see *Thread safety* in the User's Manual and the *Thread Safety Model* in the API reference.

10.3.3 Generate a single C/C++ output file, simplifying code management

A new command-line option for Code Generator, `-joinInputFiles`, joins input files and included files, processing them together to create just one output file that contains all the generated C/C++ code.

This option is useful for consolidating generated code and dependencies into a single location, especially when working with complex projects involving multiple included files. It reduces the complexity of managing imports and file dependencies, and makes it easier to distribute or deploy generated code as a standalone module.

Another new command-line option, `-joinedInputFilesOutputName <name>`, allows you to specify the output file name you want. You can optionally use this together with `-joinInputFiles`.

For details, see Command-Line Arguments for rtiddsgen in the *RTI Code Generator User's Manual*.

Note: The *RTI Code Generator User's Manual* (linked above) states that `-joinInputFiles` and `-joinedInputFilesOutputName <name>` are only valid for Python code. This is true for Code Generator in *Connext Professional*, but the *Connext TSS* version supports these command-line options for C/C++ code. Otherwise, the options function as described in the *RTI Code Generator User's Manual*.

10.3.4 FACE C APIs conform to FACE CR 1255 resolution

This release updates the FACE C APIs to match the FACE CR 1255 resolution. For more information on CR 1255, please refer to the FACE Problem Report (PR) & Change Request (CR) Ticketing System.

10.3.5 Initialize functions in generated code now comply with FACE 3.2

This release adds auto-generated inline initialize functions to C generated code in order to comply with the FACE Technical Standard, Edition 3.2.

10.3.6 Generate FACE strings and sequences in example applications

This release adds FACE strings and sequences to the `hello_goodbye` example applications generated by RTI Code Generator to help demonstrate their use. The example applications also set the connection ID variable to -1 to avoid assignment issues.

See *Getting Started* for more information on generating examples.

10.3.7 Develop Connext Professional applications for FACE

This release adds a Transport Protocol Module for *Connext Professional* 7.3.1, which allows XML-defined *Connext Professional* applications to be FACE-compliant. See *Configuring FACE 3.x for Connext Professional* for more details.

10.3.8 Use *Connext Micro* Transport Protocol Module with *Connext TSS* 4.2 architecture

This release updates the *Connext Micro* TPM to work with the new architecture of *Connext TSS* 4.2.0. This includes new templates to work with the updated version of RTI Code Generator.

10.4 What's Fixed in 4.2.0

Connext TSS 4.2.0 is a general access release based on *Connext TSS* 4.1.0.2-EAR. The following are fixes since *Connext TSS* 4.1.0.2-EAR.

10.4.1 Memory leak when using sequence-of-sequence types in General Purpose profile

When using IDL types that contained a sequence of sequences (e.g., `sequence<sequence<long>>`), a memory leak occured on the subscriber side during deserialization.

[RTI Issue ID FACETSS-1426]

10.4.2 Missing or incorrect API documentation

This issue was fixed in 4.1.0-EAR, but not documented at that time.

Some content was missing, or incorrectly located, in the C and C++ API documentation.

[RTI Issue ID FACETSS-1490]

10.4.3 Static tables that allocate memory for hash tables were determined at compile time

The size of the internal hash tables was defined at compile time and could not be changed at runtime.

[RTI Issue ID FACETSS-1749]

10.4.4 Examples caused compiler warnings on Windows systems

When building examples on Windows systems, the VS compiler warned that some files were not generated by the output. These warnings could cause incremental build issues.

[RTI Issue ID FACETSS-1862]

10.4.5 Unnecessary callback macro included in templates

The `RTI_Callback_Implemented` macro was included in templates even though it was not required.

[RTI Issue ID FACETSS-1878]

10.4.6 Incorrect error handling within create_subscriber() and create_publisher() functions

Occasionally, a remote participant would not be discovered before either `create_subscriber()` or `create_publisher()` was called from `Channel_Manager.c`. This caused a failure within the corresponding `RTI_TSS_DiscoveryConfigPlugin_assert_*` function. Since this situation is expected on occasion, it is now handled as a warning rather than a failure.

[RTI Issue ID FACETSS-1924]

10.4.7 FACE NO_ERROR collided with NO_ERROR definition on Windows systems

The FACE `NO_ERROR` enumeration collided with the `NO_ERROR` definition on Windows systems.

[RTI Issue ID FACETSS-1952]

10.4.8 Generated C code did not support forward delcarations

Generated C code did not always generate IDF type definitions (typedefs) correctly.

[RTI Issue ID FACETSS-1994]

10.4.9 Application discovery database corrupted by repetitive function calls

Connext TSS called the DPSE functions `assert_remote_participants`, `assert_remote_publications`, and `assert_remote_subscriptions` each time a new connection was created, although they should only be called once. This could corrupt the *Connext TSS* application discovery database.

[RTI Issue ID FACETSS-2028]

10.4.10 Register_Callback function signature differed from CTS GSL in C++

The `Register_Callback` function did not align with the Conformance Tool Suite (CTS) Golden Suite Library (GSL) in C++.

[RTI Issue ID FACETSS-2036]

10.4.11 Increasing sequence length caused deserialization failures in SafetyBase profile

In SafetyBase builds, capacity was not set properly due to a missing guard.

[RTI Issue ID FACETSS-2051]

10.4.12 Generated code attempted to call the private default constructor upon instantiation of an enum class

In generated code, instantiation of an enum class would attempt to call the private default constructor, due to the FACE definition of C++ enums deferring from those of DDS.

[RTI Issue ID FACETSS-2053]

10.4.13 RTI Code Generator replaced DM with TSS outside of namespaces

RTI Code Generator would occasionally replace `DM` with `TSS` outside of proper namespaces.

[RTI Issue ID FACETSS-2058]

10.4.14 Common_Hash.h generated compiler warning

Common_Hash.h generated a compiler warning related to truth value.

[RTI Issue ID FACETSS-2059]

10.4.15 return_Code in Linux example compared return value instead of assigning it

The return_code function in TSSConfigInterfaceXML.c in the *Connext Professional* Linux example compared the return value instead of setting it.

[RTI Issue ID FACETSS-2061]

10.4.16 Connext Professional C example generated compiler warnings

The C example for Connext Professional generated warnings when compiled.

[RTI Issue ID FACETSS-2062]

10.4.17 return_Code in Windows example compared return value instead of assigning it

The return_code function in TSSConfigInterfaceXML.c in the *Connext Professional* Windows example compared the return value instead of setting it.

[RTI Issue ID FACETSS-2085]

10.4.18 Incorrect library name description and example code paths in User's Manual

The example code paths in *Getting Started* did not match the actual paths in the release version.

[RTI Issue ID FACETSS-2147]

10.4.19 Possible memory leak when assigning values to sequences of strings

Connext TSS could leak memory when destroying or assigning values to a sequence of strings.

[RTI Issue ID FACETSS-2172]

10.5 Previous Releases

10.5.1 What's Fixed in 4.1.0.2-EAR

Connext TSS 4.1.0.2-EAR is a patch release based on *Connext TSS* 4.1.0.1-EAR. The following are fixes since *Connext TSS* 4.1.0.1-EAR.

Incorrect symbol used for the pre-processor #define in the generated TypeAbstraction_Callback functions

The generated TypeAbstraction_Callback headers included an extra 'l' character inserted after the 'k' in Callback. The #ifndef was not checking the correct symbol which allows the type abstraction callback header file to be included more than once. This resulted in symbol redefinition errors at compile.

[RTI Issue ID FACETSS-2013]

TSSConfiguration::Initialize() did not return on failure to malloc in examples

In the file `TSSConfigInterfaceXML.cpp` under `C++_Examples/`, the function `void TSSConfiguration::Initialize()` did not return an error code if malloc failed.

[RTI Issue ID FACETSS-2012]

Multiple nested IDL includes could generate incorrect include paths

When multiple nested includes were used in an IDL, the code generated by *rtiddsgen* may have contained the wrong include paths.

[RTI Issue ID FACETSS-1962]

C++ sequences "=" operator allocated extra memory

Previously, C++ sequences called C sequences internally. Therefore, when passing the C++ sequence with an = operator, *Connext TSS* would create a new instance and allocate memory on each pass by reference. Now, C++ sequences no longer call the C sequences internally and do not perform additional memory allocation.

[RTI Issue ID FACETSS-1963]

Connext TSS re-asserted remote publications and subscriptions for each CreateConnection() call

Connext TSS would re-assert publishers and subscribers each time `CreateConnection()` was called. *Connext TSS* now only asserts on the first call.

[RTI Issue ID FACETSS-1964]

Error returned if create_publisher() or create_subscriber() returned a warning

Connext TSS returned an error from `CreateConnection()` if `create_publisher()` or `create_subscriber()` from *Connext Micro* returned a non-significant warning. *Connext TSS* now returns a warning instead of an error.

[RTI Issue ID FACETSS-1965]

Type-specific TypeAbstraction_callback not generated properly

The `TypeAbstraction_callback` header and implementation templates were previously left blank. They now contain the correct template code.

[RTI Issue ID FACETSS-1968]

Connext TSS did not generate FACE type definitions correctly

Given certain IDL files, *Connext TSS* would incorrectly generate the required FACE type definition.

Note: If your application has any IDL files that contain a forward declaration, you should regenerate them.

[RTI Issue ID FACETSS-1978]

Connext TSS generated TypedTS includes by namespace instead of file path

`TypedTS` would use the namespace of a type instead of its file path when generating include paths.

[RTI Issue ID FACETSS-1980]

Generated code contained extra "&" when assigning internal helper functions

Generated code contained an extra & when assigning an `extra_ops` struct to RTI strings or sequences.

[RTI Issue ID FACETSS-1958]

Sequences of sequences or strings did not initialize inner objects

Sequences of sequences or strings did not properly initialize their internal objects in templates.

[RTI Issue ID FACETSS-1959]

C++ TypedTS class template did not inherit from TA_Injectable

The C++ `TypedTS` class template did not inherit properties and behaviors from the `TA_Injectable` class as intended.

[RTI Issue ID FACETSS-1967]

Generated C code did not support forward declarations

Generated C code did not always generate IDL type definitions correctly.

Note: If your application has any IDL files that contain a forward declaration, you should regenerate them.

[RTI Issue ID FACETSS-1994]

10.5.2 What's Fixed in 4.1.0.1-EAR

Connext TSS 4.1.0.1-EAR is a patch release based on *Connext TSS* 4.1.0-EAR. The following are fixes since *Connext TSS* 4.1.0-EAR.

Incorrect symbol used for the pre-processor #define in the generated TypeAbstraction_Callback functions

The generated TypeAbstraction_Callback headers included an extra 'l' character inserted after the 'k' in Callback. The #ifndef was not checking the correct symbol which allows the type abstraction callback header file to be included more than once. This resulted in symbol redefinition errors at compile.

[RTI Issue ID FACETSS-2013]

TSSConfiguration::Initialize() did not return on failure to malloc in examples

In the file `TSSConfigInterfaceXML.cpp` under `C++_Examples/`, the function `void TSSConfiguration::Initialize()` did not return an error code if malloc failed.

[RTI Issue ID FACETSS-2012]

Multiple nested IDL includes could generate incorrect include paths

When multiple nested includes were used in an IDL, the code generated by *rtiddsgen* may have contained the wrong include paths.

[RTI Issue ID FACETSS-1962]

C++ sequences "=" operator allocated extra memory

Previously, C++ sequences called C sequences internally. Therefore, when passing the C++ sequence with an = operator, *Connext TSS* would create a new instance and allocate memory on each pass by reference. Now, C++ sequences no longer call the C sequences internally and do not perform additional memory allocation.

[RTI Issue ID FACETSS-1963]

Connext TSS re-asserted remote publications and subscriptions for each CreateConnection() call

Connext TSS would re-assert publishers and subscribers each time `CreateConnection()` was called. *Connext TSS* now only asserts on the first call.

[RTI Issue ID FACETSS-1964]

Error returned if create_publisher() or create_subscriber() returned a warning

Connext TSS returned an error from `CreateConnection()` if `create_publisher()` or `create_subscriber()` from *Connext Micro* returned a non-significant warning. *Connext TSS* now returns a warning instead of an error.

[RTI Issue ID FACETSS-1965]

Type-specific TypeAbstraction_callback not generated properly

The `TypeAbstraction_callback` header and implementation templates were previously left blank. They now contain the correct template code.

[RTI Issue ID FACETSS-1968]

Connext TSS did not generate FACE type definitions correctly

Given certain IDL files, *Connext TSS* would incorrectly generate the required FACE type definition.

Note: Windows is not a FACE-supported operating system; some changes were required to get *Connext TSS* to run. Those changes are not guaranteed to be compatible with other UoCs or UoPs ported to Windows systems. Some integration may be required.

10.5.3 What's New in 4.1.0-EAR

Added support for FACE TSS C API

This release adds support for the FACE TSS C API.

Added support for Transport Protocol Module

Support for the FACE Transport Protocol Module (TPM) has been added. This feature has been implemented in the TSS/TA and as an adapter that works with *Connext Micro*. Therefore, it's an option to use both RTI's Connext TSS and TPM with *Connext Micro*. Or, either individually with other versions of TSS or TPM enabled protocols. In addition, RTI's TSS supports multiple TPMs simultaneously.

Implementing this feature required a re-architecture of the *Connext TSS*. It now implements a FACE Type Abstraction (TA) layer. In addition, this release now supports serialize and deserialize injectables.

Note: The re-architecture required to implement TPM and other features in the 4.1.0-EAR changed the way *Connext TSS* sets up and sends data with DDS. At the time of this release, 4.1.0-EAR interoperability with FACE applications using Connext TSS 3.1.2 and 3.1.1 is not tested.

Implemented certifiable FACE String and Sequence classes

This release implements the FACE string and sequence classes in C and wraps them with an optional C++ API. This new implementation is easier to certify and addresses the following issues reported in previous versions:

- Freeing and reallocating string and sequence classes did not work in Safety Base.
- Destructor of sequences of sequences created a memory leak.
- The C++ sequence class required casting.
- Bound, Capacity, and Length of sequences of structures became corrupted.
- Having two sequence types in the same structure could crash the application.

RTI logging replaced by FACE logging

RTI's proprietary logging has been replaced with the FACE logging injectable. For more information, see the *Logging* section of the API reference.

Added support for LynxOS-178

Support for the LynxOS-178 ARINC operating system has been added.

Added support for FACE GUIDs

To increase FACE conformance, *Connext TSS* now supports FACE GUIDs.

Added injectable interface for callbacks

Added an injectable interface for callbacks.

Third-party software changes

The following third-party software is now used by *Connext TSS*:

Table 10.3: New Third-Party Software

Third-Party Software	Version
XML Parser for C	0.1.0

For information on all third-party software used by *Connext TSS*, see the *Third-Party Software* document in this manual.

10.5.4 What's Fixed in 4.1.0-EAR

Sequences of arrays caused compile errors

The managed copy constructor of a sequence of arrays attempted to copy the contained arrays incorrectly.

[RTI Issue ID FACETSS-1296]

Possible memory leak when RTI_TSS_Impl_new() did not check for NULL after allocation

If memory was unavailable, a null pointer could have been passed to an underlying DDS call when creating a new TSS instance.

[RTI Issue ID FACETSS-1279]

No longer necessary to use #ifdef FACE_SEQUENCE_AND_STRING_IMPLEMENTED flag

The `FACE_SEQUENCE_AND_STRING_IMPLEMENTED` flag has been removed. Therefore, it's not necessary to use this flag to enable strings and sequences.

[RTI Issue ID FACETSS-1370]

10.5.5 What's New in 3.1.2

Added support for Deos 653

This release adds support for building the TSS libraries on the DDC-I DESK Console, as well as an example to be built and run on a Deos 653 CPU via OpenArbor.

Changed Logging verbosity to be set by command line switches

In this release, logging verbosity is changed by command line switches passed to Cmake while building the TSS Libraries. See the *Building the Libraries* section of the *Building Connext TSS* chapter for more information.

Note: Logging verbosity can only be set at compile time, not at run time.

Renamed Cmake architecture files for VxWorks7 to include FACE_GP

The cmake architecture files `armv8Vx7SR0660llvm10.0.1.cortex-a53.cmake` and `armv8Vx7SR0660llvm10.0.1.cortex-a53_rtp.cmake` have been renamed to `armv8Vx7SR0660llvm10.0.1.cortex-a53FACE_GP.cmake` and `armv8Vx7SR0660llvm10.0.1.cortex-a53_rtpFACE_GP.cmake`, respectively.

10.5.6 What's Fixed in 3.1.2

Compiler warnings on VxWorks 7 systems

When building for VxWorks 7 systems with the llvm 10.0.1 compiler (`armv8Vx7SR0660llvm10.0.1.cortex-a53_rtp_pro_Release`), a `CPU macro redefined` compile warning could occur if `-Wmacro-redefined` was enabled on the compiler command line. You may have also seen a `-Wreturn-type-c-linkage` warning.

[RTI Issue ID FACETSS-955]

Warning when loading hello_goodbye_dkm_static example

The following warning occured when loading the `hello_goodbye_dkm_static` example into a physical board:

```
ld 1< hello_goodbye_app.so
Warning: module 0xffff800000aaab60 holds reference to undefined symbol __dso_handle.
ld(): module contains undefined symbol(s) and may be unusable.
value = 0 = 0x0
```

[RTI Issue ID FACETSS-1048]

Build time FACE conformance warning message

In previous releases, when building *Connext TSS* with *Connext Professional* for the FACE GeneralPurpose profile, the build system generated the following message:

```
-- Connext TSS with Connext DDS Pro does not conform with FACE GeneralPurpose Profile␣
↪in this version.
```

This message was not accurate and is no longer generated.

[RTI Issue ID FACETSS-1086]

Double quotation marks in command line contexts

In the Porting documentation for this release, instances of double quotation marks (“) in command line contexts have been changed to double quotes (”) to avoid corrupting OpenArbor projects with copied and pasted commands.

[RTI Issue ID FACETSS-729]

Source file types.c was compiled when flag was set to true

In previous releases, the file `src/micro/types.c` was still compiled and linked with the *Connext TSS* library when `FACE_SEQUENCE_AND_STRING_IMPLEMENTED` was set to false. This issue has been resolved.

[RTI Issue ID FACETSS-905]

Receive_Message() unable to retrieve multiple messages in the reader queue

When a TSS application called `Receive_Message()` while multiple messages were present in the reader queue, the next `Receive_Message()` call would block on the waitset until a new sample arrived or the timeout expired. This issue has been resolved.

[RTI Issue ID FACETSS-679]

10.5.7 What's New in 3.1.1

Removed the Static Configuration API

Static configuration has been removed from the TSS code and generated files. Some header files have been removed, hence, this release is incompatible with previous engineering releases. There are changes to the generated files as well; code regeneration is required for this release.

Previous engineering releases included a note that this was being deprecated. Now it is removed.

Updated source files to meet RTI and FACE coding standards

The source files have been reformatted to meet the RTI and FACE coding standards.

Updated parameter checking and return codes

This release includes updated parameter checking and the associated return codes for the SafetyBase profile.

10.5.8 What's Fixed in 3.1.1

Updated rtiddsgen to meet FACE 3.1 Conformance

A bug in rtiddsgen caused the TypedTS and TypedTS Injectable files to not be generated in the folder structure defined in the spec.

This issue has been resolved. The TypedTS and TypedTS Injectable files will now be created in the correct folder structure.

[RTI Issue ID FACETSS-641]

11 Third-Party Software

This section outlines RTI's use of third-party and open source software in *Connext TSS*.

11.1 Third-Party Software included in Connext TSS

11.1.1 XML Parser for C

- Used in the some of the shipped examples as a way to parse XML input.
- Version: 0.1.0
- Source: https://github.com/recp/xml/releases/tag/v0.1.0/
- License:

```
MIT License

Copyright (c) 2019 Recep Aslantas

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

11.2 Third-Party Software Used by RTIDDSGEN Code-Generation Utility

You can also find the complete Software Bill of Materials (SBOM) in the <installation_directory>/sbom folder. RTI ships the SBOM so you can process them using your usual cybersecurity and software management tools. The SBOM includes all dependencies, not just first-level ones; therefore, it contains the complete list of third-party software that RTI uses to build the *Connext* libraries, infrastructure services, and tools.

11.2.1 ANTLR

- This software is distributed with rtiddsgen (RTI Code Generator) as a jar file. The source code is not modified or shipped. In addition, the output produced by this software from a grammar file is part of the rtiddsgen JAR file.
- Version: Release 3.5.2
- License: https://www.antlr3.org/license.html

```
[The BSD License]

Copyright (c) 2010 Terence Parr

All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

Neither the name of the author nor the names of its contributors may
be used to endorse or promote products derived from this software
without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

11.2.2 Apache Commons CLI

- Version 1.5.0
- This software is distributed with rtiddsgen (RTI Code Generator) as a jar file. The source code is not modified or shipped.
- License: Apache License Version 2.0 (full text found in *Open Source Software Licenses*).

11.2.3 Apache Commons Lang

- Used on Code Generator. We only use the class StringUtils.
- Version 2.6
- License: Apache License Version 2.0 (full text found in *Open Source Software Licenses*).

11.2.4 Apache Log4j 2

- This software is distributed with rtiddsgen (RTI Code Generator) as a jar file. The source code is not modified or shipped.
- Source: https://www.apache.org/licenses/LICENSE-2.0
- Version: 2.17.1

11.2.5 Apache Velocity

- This software is included in rtiddsgen (RTI Code Generator). The source code is not modified or shipped.
- Source: http://velocity.apache.org/
- Version: 2.3
- License: The Apache Software License, Version 2.0
  http://velocity.apache.org/engine/devel/license.html

11.2.6 Gson

- Version 2.9.1
- Portions of rtiddsgen (RTI Code Generator) are built using Gson.
- License: The Apache Software License, Version 2.0 (full text found in *Open Source Software Licenses*)

11.3 Open Source Software Licenses

11.3.1 Apache License version 2.0, January 2004 (http://www.apache.org/licenses/)

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.
```

(continues on next page)

(continued from previous page)

```
"License" shall mean the terms and conditions for use, reproduction,
and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by
the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all
other entities that control, are controlled by, or are under common
control with that entity. For the purposes of this definition,
"control" means (i) the power, direct or indirect, to cause the
direction or management of such entity, whether by contract or
otherwise, or (ii) ownership of fifty percent (50%) or more of the
outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity
exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications,
including but not limited to software source code, documentation
source, and configuration files.

"Object" form shall mean any form resulting from mechanical
transformation or translation of a Source form, including but
not limited to compiled object code, generated documentation,
and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or
Object form, made available under the License, as indicated by a
copyright notice that is included in or attached to the work
(an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object
form, that is based on (or derived from) the Work and for which the
editorial revisions, annotations, elaborations, or other modifications
represent, as a whole, an original work of authorship. For the purposes
of this License, Derivative Works shall not include works that remain
separable from, or merely link (or bind by name) to the interfaces of,
the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including
the original version of the Work and any modifications or additions
to that Work or Derivative Works thereof, that is intentionally
submitted to Licensor for inclusion in the Work by the copyright owner
or by an individual or Legal Entity authorized to submit on behalf of
the copyright owner. For the purposes of this definition, "submitted"
means any form of electronic, verbal, or written communication sent
to the Licensor or its representatives, including but not limited to
communication on electronic mailing lists, source code control systems,
and issue tracking systems that are managed by, or on behalf of, the
Licensor for the purpose of discussing and improving the Work, but
excluding communication that is conspicuously marked or otherwise
designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity
on behalf of whom a Contribution has been received by Licensor and
subsequently incorporated within the Work.
```

(continues on next page)

(continued from previous page)

```
2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
```

(continues on next page)

(continued from previous page)

```
        as modifying the License.

    You may add Your own copyright statement to Your modifications and
    may provide additional or different license terms and conditions
    for use, reproduction, or distribution of Your modifications, or
    for any such Derivative Works as a whole, provided Your use,
    reproduction, and distribution of the Work otherwise complies with
    the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
    any Contribution intentionally submitted for inclusion in the Work
    by You to the Licensor shall be under the terms and conditions of
    this License, without any additional terms or conditions.
    Notwithstanding the above, nothing herein shall supersede or modify
    the terms of any separate license agreement you may have executed
    with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
    names, trademarks, service marks, or product names of the Licensor,
    except as required for reasonable and customary use in describing the
    origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
    agreed to in writing, Licensor provides the Work (and each
    Contributor provides its Contributions) on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied, including, without limitation, any warranties or conditions
    of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
    PARTICULAR PURPOSE. You are solely responsible for determining the
    appropriateness of using or redistributing the Work and assume any
    risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
    whether in tort (including negligence), contract, or otherwise,
    unless required by applicable law (such as deliberate and grossly
    negligent acts) or agreed to in writing, shall any Contributor be
    liable to You for damages, including any direct, indirect, special,
    incidental, or consequential damages of any character arising as a
    result of this License or out of the use or inability to use the
    Work (including but not limited to damages for loss of goodwill,
    work stoppage, computer failure or malfunction, or any and all
    other commercial damages or losses), even if such Contributor
    has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
    the Work or Derivative Works thereof, You may choose to offer,
    and charge a fee for, acceptance of support, warranty, indemnity,
    or other liability obligations and/or rights consistent with this
    License. However, in accepting such obligations, You may act only
    on Your own behalf and on Your sole responsibility, not on behalf
    of any other Contributor, and only if You agree to indemnify,
    defend, and hold each Contributor harmless for any liability
    incurred by, or claims asserted against, such Contributor by reason
    of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

(continues on next page)

(continued from previous page)

```
APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

12 Copyrights and Notices

Trademarks

Copy and Use Restrictions

Third-Party software

Notices

AI Use Disclosure

Our engineering team uses AI-assisted tools to accelerate code generation and debugging. However, human developers retain full ownership; every line of code is reviewed and approved by human developers.

Deprecations and Removals

Deprecated means that the item is still supported in this release, but will be removed in a future release. *Removed* means that the item is discontinued or no longer supported.

Any deprecations or removals noted in RTI's documentation serve as notice under the Real-Time Innovations, Inc., Maintenance Policy #4220 and/or any other agreements by and between RTI and customer regarding maintenance

and support of RTI's software. RTI's current standard terms and support and maintenance policies are available at https://www.rti.com/terms.

Technical Support Real-Time Innovations, Inc. 232 E. Java Drive Sunnyvale, CA 94089 Phone: (408) 990-7444 Email: support@rti.com Website: https://support.rti.com/