



TASKING RTE for TriCore/PPU User Guide

Copyright © 2025 TASKING B.V.

All rights reserved. You are permitted to print this document provided that (1) the use of such is for personal use only and will not be copied or posted on any network computer or broadcast in any media, and (2) no modifications of the document is made. Unauthorized duplication, in whole or part, of this document by any means, mechanical or electronic, including translation into another language, except for brief excerpts in published reviews, is prohibited without the express written permission of TASKING B.V. Unauthorized duplication of this work may also be prohibited by local statute. Violators may be subject to both criminal and civil penalties, including fines and/or imprisonment. TASKING® and its logo are registered trademarks of TASKING Germany GmbH. All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed.

Table of Contents

Manual Purpose and Structure	v
1. Overview of the Run-time Environment	1
1.1. Introduction	1
1.2. Glossary	1
1.3. Features	2
1.3.1. Support for User Defined DPFs	2
1.3.2. Synchronous Execution of DPFs	2
1.3.3. Asynchronous Execution of DPFs	2
1.3.4. Multi-Core Support	3
1.3.5. Server Side RTE: A Bare Metal Application	3
1.3.6. AUTOSAR Complex Device Driver Integration	3
1.4. Overview Bare-Bones RTE	3
1.5. Overview AUTOSAR	8
1.6. Overview AUTOSAR CDD	9
2. Using the Run-time Environment	11
2.1. Unpacking the Run-time Environment	11
2.2. Using the RTE Multi-Core Demo in SmartCode Eclipse	11
2.3. Using the RTE Standalone with CMake	14
2.4. Baremetal Instructions	15
2.4.1. Adding Custom Data Processing Function	16
2.4.2. Adding More Jobs and Increasing Number of Operations	16
2.4.3. Adding More Queues and Increasing Their Sizes	16
2.4.4. Adding More Memory Partitions	17
2.4.5. Setting Up the Interrupts Used by Notification Method Callback	17
2.4.6. RTE Startup Core	17
2.5. Using RTE in AUTOSAR	17
2.5.1. Installation/Integration	17
2.5.2. CDD_PpuArLayer Functionality	18
2.5.3. RTE AUTOSAR Example	19
2.6. Using the RTE Configuration Tool	20

Manual Purpose and Structure

Manual Purpose

You should read this manual if you want to know how to use the TASKING RTE for TriCore/PPU with TASKING SmartCode.

The TASKING RTE for TriCore/PPU consists of a TriCore component and a Parallel Processing Unit (PPU) component and supports the execution of data processing functions (DPF) on the PPU vector core upon request from applications running on a TriCore core.

Please note that you need to use this product with TASKING SmartCode v10.4r1.

Manual Structure

Chapter 1, Overview of the Run-time Environment

Contains an introduction to the TASKING RTE for TriCore/PPU and an overview about the product components.

Chapter 2, Using the Run-time Environment

Explains how to use the different parts of the TASKING RTE for TriCore/PPU with TASKING SmartCode.

Related Publications

- Getting Started with TASKING SmartCode
- TASKING SmartCode - TriCore User Guide
- TASKING SmartCode - ARC/PPU User Guide

Chapter 1. Overview of the Run-time Environment

1.1. Introduction

The TASKING RTE for TriCore/PPU is a run-time environment to exchange data between the TriCore and the PPU. The so-called Bare-Bones RTE uses the PPU either to offload the TriCore or to speedup calculations by using the vector processing capabilities (Vector DSP Unit) of the PPU. Due to its parallelism, operations on data objects can potentially be executed much faster on the PPU than on a TriCore. However, to use the PPU, the data is first transferred to the local PPU vector memory (VCCM). After the vectorized operations on the data, the results can be transferred back from the PPU vector memory to a TriCore memory such as LMU (0-9), DLMU (0-5) or DSPR (0-5).

The RTE product contains the following:

- Client side RTE in the form of an AUTOSAR Complex Device Driver (CDD) with source code and configuration files
- Example projects using RTE services
- This user guide

1.2. Glossary

The following terms and abbreviations are used in this manual.

Abbreviation	Description
AUTOSAR	AUTomotive Open System ARchitecture - a global partnership of leading companies in the automotive and software industry to develop and establish the standardized software framework and open E/E system architecture for intelligent mobility.
AUTOSAR OS	AUTomotive Open System ARchitecture Operating System - the operating system of the AUTOSAR environment mainly offering the tasks (cyclic and/or non-cyclic) where all the AUTOSAR module functions are called.
AUTOSAR RTE	AUTomotive Open System ARchitecture Run-Time Environment - acts as a middleware between the AUTOSAR application layer and the lower layers. Basically, the RTE layer manages the inter- and intra-ECU communication between application layer components as well as between the BSW and the application layer.
BSW	Basic SoftWare - can be defined as standardized software module offering various services necessary to run the functional part of the upper software layer. This layer consists of the ECU specific modules along with the generic AUTOSAR modules.
CDD	Complex Device Driver - used to be the acronym for Complex Device Driver or Complex Driver, but is not limited to drivers.
CSM	Cluster Shared Memory

Abbreviation	Description
DLMU	Data Local Memory Unit
DPF	Data Processing Function
DSPR	Data Scratch-Pad RAM
DTF	Data Transfer Function
ECU	Electronic Control Unit
LMU	Local Memory Unit
MCAL	MicroController Abstraction Layer - the AUTOSAR compliant software module that has direct access to all the on-chip MCU peripheral modules and external devices, which are mapped to memory. It makes the upper software layers independent of the MCU.
MCU	MicroController Unit
Operation	a DTF or DPF
PPU	Parallel Processing Unit
RTE	Run-Time Environment
STU	Streaming Transfer Unit
SWC	SoftWare Component - the AUTOSAR Application layer constitutes the topmost layer within an AUTOSAR software architecture and is identified to be critical for all the vehicle applications. The AUTOSAR standard specifies the application layer implementation using a "component" concept.
VMEM/VCCM	Vector Memory/Vector Closely Coupled Memory

1.3. Features

1.3.1. Support for User Defined DPFs

The RTE supports user defined Data Processing Functions (DPFs) with an arbitrary number of parameters.

1.3.2. Synchronous Execution of DPFs

The client side RTE supports synchronous execution of DPF requests, where the Run-Time Library (RTL) function waits for the job completion. The return value will indicate the success or error condition of the job. On success, the output data of the DPF is available at the requested memory locations. However, in case of failure, the contents of the output data are unspecified.

1.3.3. Asynchronous Execution of DPFs

The client side RTE supports asynchronous execution of DPF requests, where the Run-Time Library (RTL) function call returns with a confirmation that the request is submitted to the server side. Once the server side processes the request, the RTE notifies the client side application, in case of successful completion the output data of the DPF is available at the requested memory locations. However, if there are any failures in client side or server side, the necessary return code or notification is provided.

Due to the asynchronous DPF request execution, subsequent requests may be submitted before an earlier request is completed. These requests are queued on the server side, ensuring simplicity on the client side. However, it implies that the client side RTL functions are reentrant. The RTL provides functions to inquire about a request's status and to cancel requests.

Please note that the maximum size of a request queue can be configured at compile-time.

1.3.4. Multi-Core Support

Multiple processor core support can be achieved by having a job queue for each core. If a single job queue is to be used by multiple cores (or tasks, in case of an RTOS), you must protect the queue by a locking mechanism to prevent concurrent access to the queue.

1.3.5. Server Side RTE: A Bare Metal Application

The server side of the run-time environment is a bare metal application which consists of initialization of the RTE for the PPU side and a forever while loop processing the jobs in the available job queues. The initialization of the PPU server side consists of initialization of the Streaming Transfer Unit (STU), initialization of data in Cluster Shared Memory (CSM) and Vector Core Coupled Memory (VCCM), and signaling readiness. It also handles exceptions that occur when data processing requests are executed by informing the client side and by returning to a state in which it can accept new requests.

1.3.6. AUTOSAR Complex Device Driver Integration

The classic AUTOSAR Complex Device Driver (CDD) provides the client side run-time support for PPU interfacing. The RTE provides the interfaces required by AUTOSAR for a CDD, including their formal ARXML definitions. This involves utilizing specific services provided by an AUTOSAR run-time environment, to the extent that these services can replace those provided by the standalone client-side RTE.

1.4. Overview Bare-Bones RTE

The goal of the Bare-Bones RTE is to use the PPU to either offload the TriCore or to speedup calculations by using the Vector processing capabilities (Vector DSP Unit) of the PPU.

The minimal configuration for the RTE consists of a queue and a job. A job consists of a number of operations. Operations can be memory transfers and/or Data Processing Functions (DPFs). Memory transfers are for transferring data between memories. This most likely is between TriCore memory and PPU memories (VCCM or CSM). However, you can implement any transfer as long as the Streaming Transfer Unit (STU) of the PPU has access to the memories. After memory transfer a DPF (which is within the job also an operation) can operate on this data, after which the result can be transferred back to TriCore or used as input for an other DPF. Memory transfer operations are performed by the Streaming Transfer Unit (STU) of the PPU. A predefined memory transfer operation is available for both 1D and 2D addressing modes.

Note that in the `cstart` configuration `cstart.h`, the STU must be given write access to the memories that contain destination buffers for the read and read_2d DTFs. PPU.STU is bit 24 of the `WRB_VALUE` value. To allow write access set this bit to 1 and set the corresponding `WRB_INIT` value to 1. For example,

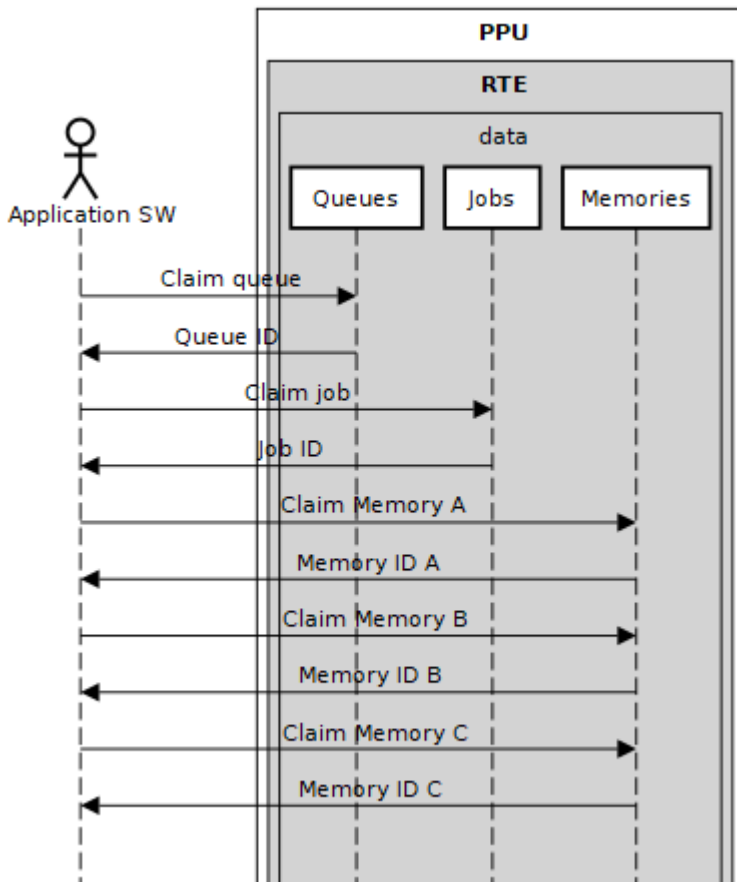
```
#define __CPU0_ACCENSPRCFG_WRB_INIT 1  
#define __CPU0_ACCENSPRCFG_WRB_VALUE 0x01000000
```

Stages

The Bare-Bones RTE can be subdivided into several stages.

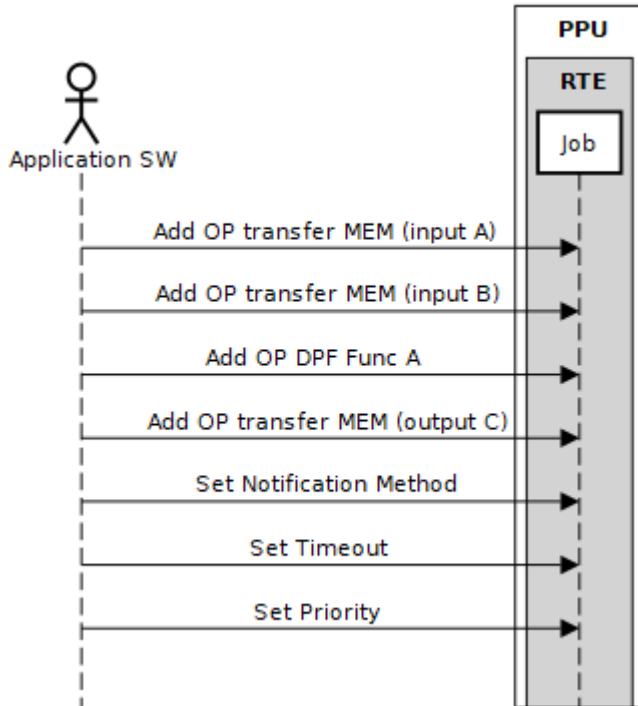
1. Claiming resources.
2. Setting up the job.
3. Execution of the job.
4. If necessary, releasing the claimed resources. Releasing the resources may not be necessary when the Job and Job Queue are continuously used e.g. when a user program periodically submits a Job to a Job Queue.

Claim Resources



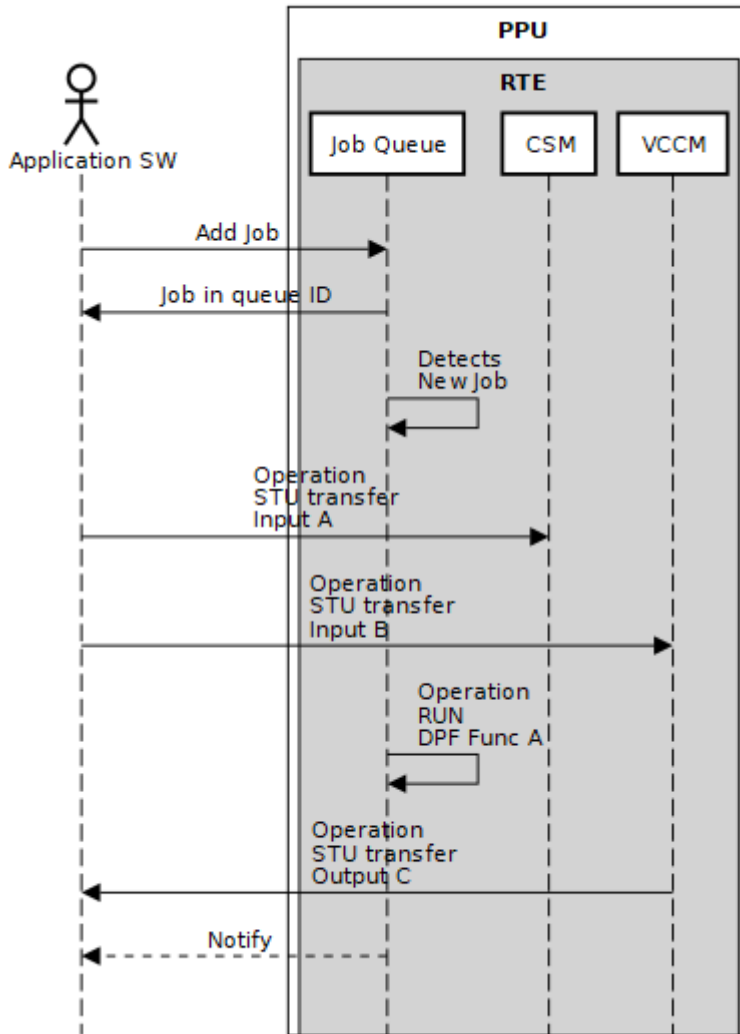
The Application SW allocates (claims) a job and a queue. Operations are added to the job, a typical job consists of memory transfer(s) for input data, data processing function and memory transfer for output data.

Prepare Job



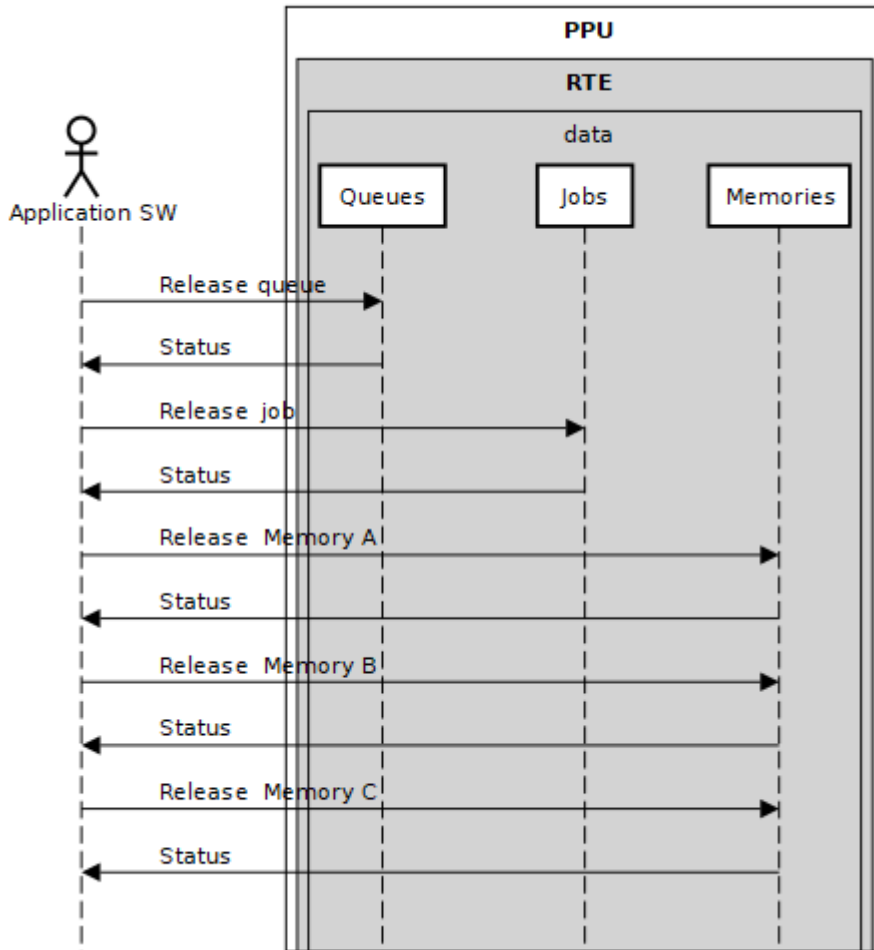
After the operations for the job are set the job can be added to the queue. The PPU side will scan the queue(s) for new jobs, sort them on priority and do the operations of the job with the highest priority. Priority 0 is the highest priority.

DPF Flow



Job completion can be handled in several ways, this is specified by the job notification method. The job notification method can be one of the following methods: none, wait, poll and callback.

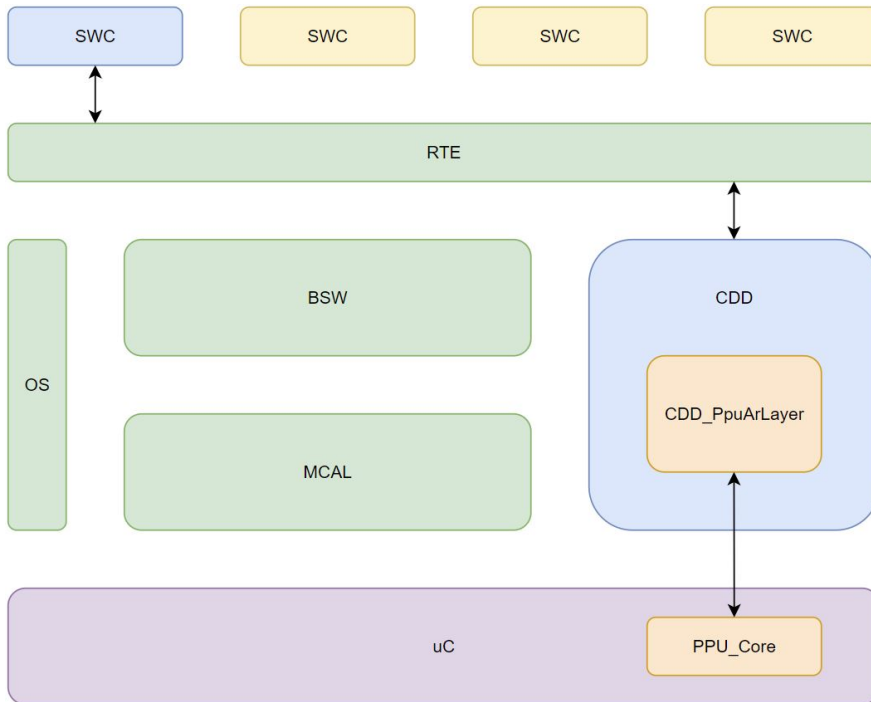
Release Resources



When the job is done the claimed resources can be released.

1.5. Overview AUTOSAR

The following figure shows a schematic overview of the RTE in an AUTOSAR configuration:



The actor in the AUTOSAR layer is the AUTOSAR Software Component (AR - SWC), a naming assigned to a software module that can run on any TriCore AUTOSAR compliant core. The SWC adds jobs via an API using the AUTOSAR RTE layer configuration from any core. The AUTOSAR RTE configuration should ensure and take care of any inter-core communication for passing the data in both synchronous and asynchronous API calls.

The AUTOSAR CDD layer gets the input data from the AUTOSAR RTE and if needed adapts the data to be used in the CDD's core APIs that pass that data to the PPU to be processed.

Optionally, the SWC can check the status of the job added in the CDD.

After the processing is done, the AUTOSAR CDD layer sets the job, task status and other data based on the processing result from the lower layers. The SWC will be able to access the results using various API provided to check status and read/write data.

1.6. Overview AUTOSAR CDD

This section describes the responsibilities of the Job Queue and how it collaborates with the AUTOSAR CDD Interface Layer.

Collaborations and responsibilities:

1. Provide a mechanism to describe input buffer transfer to the CDD Interface Layer.
2. Provide a mechanism to describe output buffer transfer to the CDD Interface Layer.
3. Provide a mechanism to execute a DPF to the CDD Interface Layer.
4. Provide a mechanism to report status to the CDD Interface Layer.

The CDD Interface Layer and Application SW are used as proxies for each other.

The Job Queue will consume Jobs from the CDD Interface Layer.

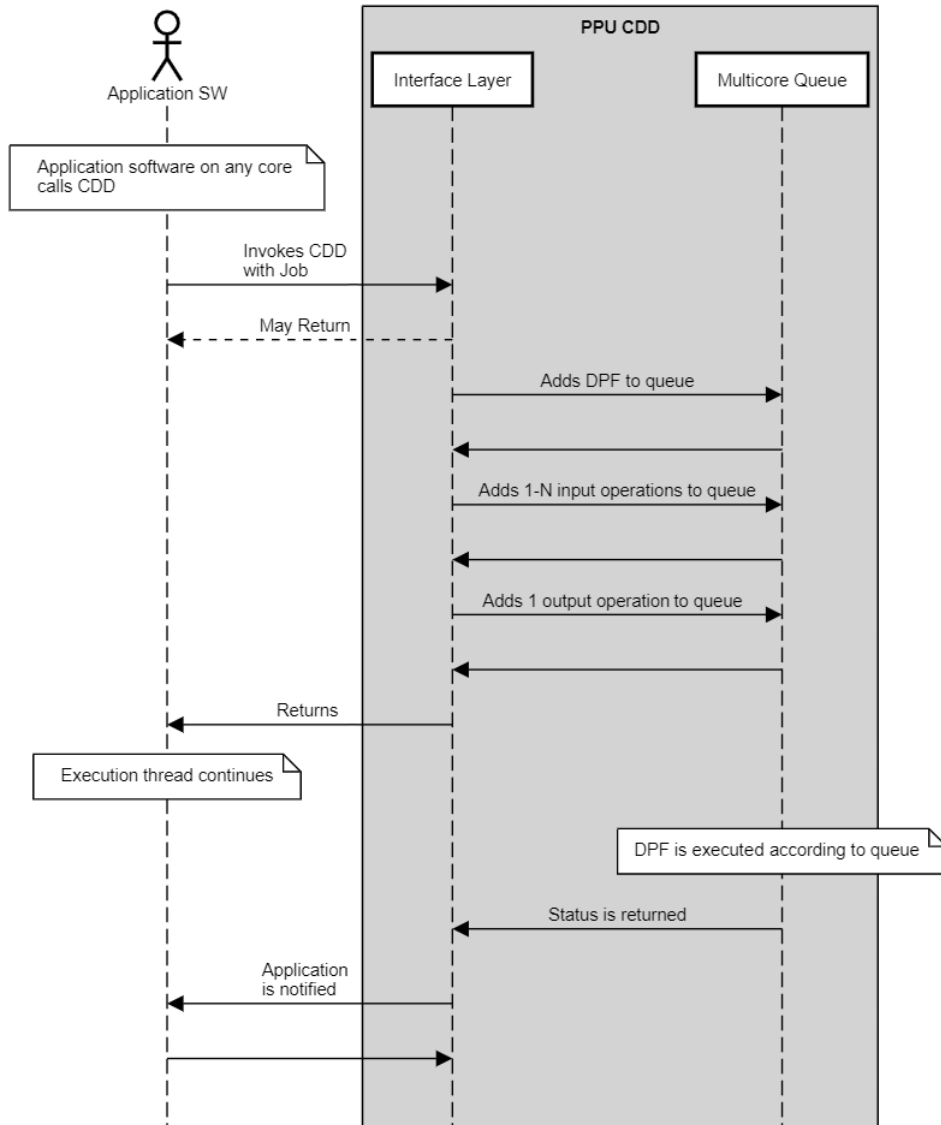
A Job consists of:

1. 0 - N buffers (in, out, in/out)
2. 0 - M parameter
3. 0 - 1 job priority (optional)
4. 1 DPF
5. job notification type (by default, the job notification is set to POLL)

The Job Queue will notify the CDD Interface Layer upon completion of the job. A job can complete in three ways: DPF done, output data transfer complete or an error condition. Job completion can be synchronous or asynchronous, with synchronous requiring an internal wait until the job is completed, whereas asynchronous allows for periodic status checks via the AUTOSAR interface to track the current state of the job.

Note that the job completion notification via a callback routine is currently not supported by the AUTOSAR CDD.

High-level Overview Multi-Core Queue



Chapter 2. Using the Run-time Environment

The libraries of the RTE product are protected by a license. A license that allows unpacking the libraries is required in order to use the RTE product.

2.1. Unpacking the Run-time Environment

The installed SmartCode product contains three unpackers for the Run-time Environment, which are located in the following directories:

- `<install-dir>/SmartCode <version>/carc/lib` (unpack-arc-libppu_rte-library)
- `<install-dir>/SmartCode <version>/ctc/bin` (unpack-tc-libppu_rte-binary)
- `<install-dir>/SmartCode <version>/ctc/lib` (unpack-tc-libppu_rte-library)

To unpack, you can run the respective unpack executable, though on Windows, you may need to run the unpackers as an administrator depending on the installation directory.

2.2. Using the RTE Multi-Core Demo in SmartCode Eclipse

You can use the `rte_multicore_demo` project in SmartCode Eclipse. It demonstrates how you can setup a multi-core RTE.

In this example all 6 cores run a matrix addition job. Each core has its own queue, its own job and its own memories that are needed for the addition job. As notification method a notify via callback is used.

To import the example in SmartCode Eclipse, perform the following steps in Eclipse.

1. From the **File** menu, select **Import...**

The Import dialog appears.

2. Expand **TASKING C/C++** and select **TASKING TriCore Example Projects » Next.**

The Import TASKING TriCore Example Projects dialog appears.

3. Select the projects `rte_ppu`, `rte_tc0`, `rte_tc1`, `rte_tc2`, `rte_tc3`, `rte_tc4`, and `rte_tc5`.

4. Click **Finish**.

The projects are now visible in the C/C++ Projects view.

5. In the C/C++ Projects view right-click on `rte_tc0` and select **Set Active Project**.

6. From the **Project** menu select **Build rte_tc0**

This will build the project. Once finished seven `.elf` files are present in the Debug directory.

TASKING RTE for TriCore/PPU User Guide

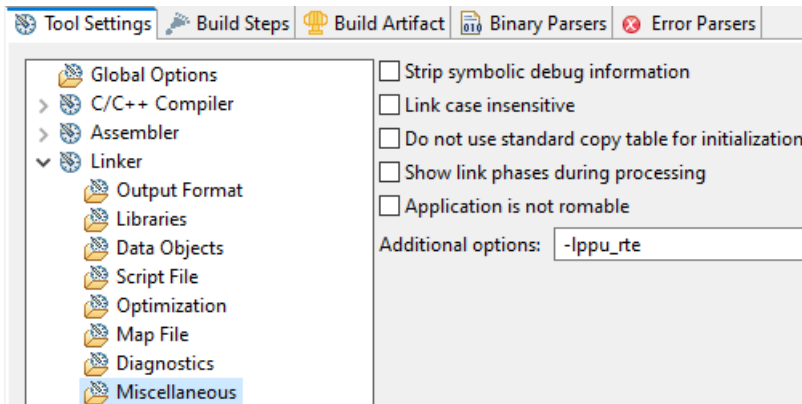
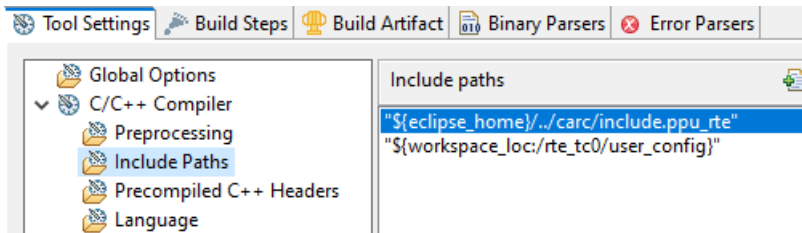
You can now debug/run the project. A winIDEA (standalone) project is provided in the `rte_tc0` project directory.

Project settings

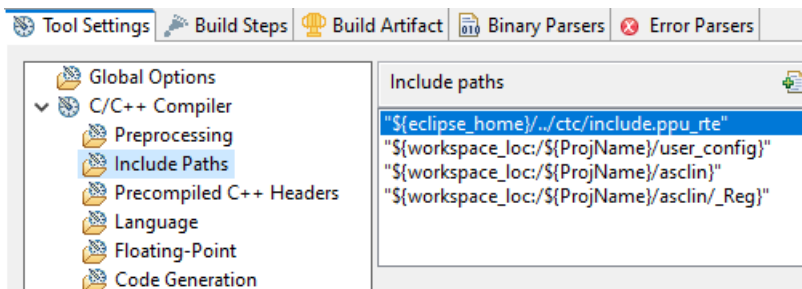
Note that when you create your own RTE project, you need to include the `include.ppu_rte` search path, and you need to link the `libppu_rte.a` library (linker option `-lppu_rte`), this is necessary for the PPU project and the `tc0` or `vtc` project.

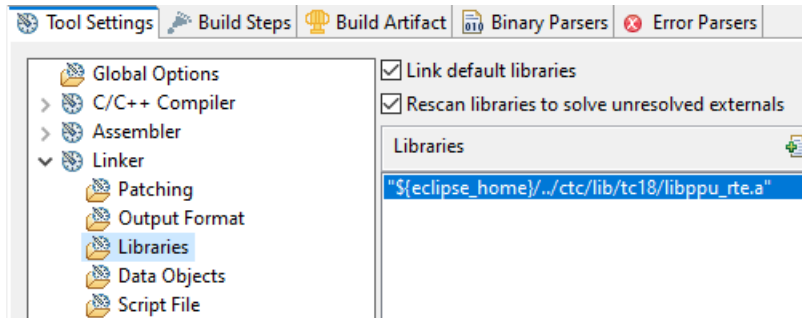
These are the project properties used in the example `rte_multicore_demo` project:

For `rte_ppu`:



For `rte_tc0`:





Working of the example

At first each core 'claims' its queue, job and memories. Then the job is configured by:

- job init
- set job notification method
- adding memory transfer for input data
- adding the add operation
- adding memory transfer for output data

The job is now ready to be used by adding it to the queue. In the example, before a job is added to the queue the job priority number is set to a random value between 0 and 15.

After adding the job to the queue, the sum of the addition is calculated locally and compared to the result of the RTE job when a job done notification is received.

The example is made for a TriCore TC4D COM board and sets up ASCLIN0 for communication with your PC. A winIDEA project file is available in the `rte_tc0` project directory.

Every five seconds the example will show how many jobs have been performed and how many errors have been encountered in the calculation comparison (the number of errors should be 0). The output will look like:

```
5.000442 seconds (uptime 00:00:55)
rte tc0 jobs 40048 (delta), errors 0 (total)
rte tc1 jobs 40208 (delta), errors 0 (total)
rte tc2 jobs 39963 (delta), errors 0 (total)
rte tc3 jobs 40020 (delta), errors 0 (total)
rte tc4 jobs 39854 (delta), errors 0 (total)
rte tc5 jobs 40043 (delta), errors 0 (total)
```

For more details on the example, see the `readme.txt` file in `examples/rte_multicore_demo/rte_tc0`.

2.3. Using the RTE Standalone with CMake

Two standalone RTE examples for use with CMake or amk are present in the directory `<SmartCode-install-dir>/ctc/examples/rte_custom_dpf_demo` and `<SmartCode-install-dir>/ctc/examples/rte_performance_demo`. The first example shows how to use the predefined DPFs and how to create your own DPF and add it to the system. Meanwhile, the second example comes with a comparison of execution times between scalar naive implementation of matrix multiplication and the vectorized implementation running within the RTE library. The vectorized implementation uses a 1x16 kernel for calculations to fully take advantage of the vector unit without being unfair to the scalar implementation.

Both examples include a preconfigured `cstart` and linker file with the correct settings for memory access.

1. To compile, make sure that the paths to `ctc/bin` and `carc/bin` are included in the executable search path for the command line.
2. Files `build.bat` (Windows) and `build.sh` (Linux) contain all necessary commands to build the project. A makefile for use with `amk` is also available for both projects.
3. Flash the generated `rte_ppu_custom_dpf_example.elf` or `rte_performance_example.elf` file located in the `build` directory of the project.

Dependencies for building the CMake example

Windows

- SmartCode v10.4r1
- Python (version at least 3.8)

For CMake build:

- CMake (version at least 3.16)
- MinGW or GNUMake

You can change the environment variable `MAKE_COMMAND` on Windows to select your preferred tool.

Linux

- SmartCode v10.4r1
- Python (version at least 3.8)

For CMake build:

- CMake (version at least 3.16)
- Make

2.4. Baremetal Instructions

The TASKING RTE uses a configuration script `tsk_rte_config_tool.py` to generate internal data structures and functions used by the PPU Run-time, according to the configuration specified by you. See [Section 2.6, Using the RTE Configuration Tool](#) for more information.

The configuration script can take several arguments, e.g. `--config-dir` which takes the path to the `user_config` directory containing configuration files, and `--vtc-dir`, `--tc0-dir`, `--tc1-dir`, `--tc2-dir`, `--tc3-dir`, `--tc4-dir`, `--tc5-dir`, `--ppu-dir`, and `--shared-dir` to specify the paths where the output files have to be created.

The script supports two types of projects. `--vtc-dir` is for when the project is a VTC project that contains all code shared between the TriCore cores, and `NO_VTC` in which each TriCore core has its own project.

Note that the shared directory should be visible (using option `-I`) to both compilation units ARC and TriCore.

Alternatively, you can use the script with the argument `--generate-default-config`. This results in the script generating a baseline configuration, with optionally the `--config-dir` option to specify the directory in which to create the default configuration files.

The configuration consists of the following three files: `user_dpf_config.h`, `user_dpf_functions.h` and `user_pools_config.h`.

The default configuration contains a few queues with an example buffer. You can use this as a starting point for creating your project with the RTE.

Example:

```
python3 tsk_rte_config_tool.py --config-dir user_config
                                --vtc-dir src/tc --ppu-dir src/ppu --shared-dir src/shared
python3 tsk_rte_config_tool.py --config-dir user_config --shared-dir ./
python3 tsk_rte_config_tool.py --generate-default-config
```

The configuration tool `tsk_rte_config_tool.py` generates the following files in the specified directories:

1. `src/tc` directory

`g_tsk_rte_init.c`, which contains an init function for establishing the internal library object pointers and critical limit definitions.

2. `src/ppu` directory

`g_tsk_rte_config_ppu.c`, which contains the definitions of the library storage elements and a DPF lookup table.

`g_tsk_rte_init_ppu.c`, which has a similar functionality to the one in TriCore, but is specifically for PPU.

3. `src/shared` directory

`g_tsk_rte_config.h`, which contains referenceable IDs for user-defined items from the configuration that match the names provided in the configuration.

`g_tsk_rte_dpf.h`, which is at the top of the hierarchy of payloads used to define the final size. The user payloads should be present here as a member of a union.

`g_tsk_rte_function_id.h`, which contains the IDs of functions for the predefined DPFs and user-defined ones.

The following sections explain the steps needed to configure the project.

2.4.1. Adding Custom Data Processing Function

Adding a custom DPF is one of the features of the RTE. In order to add a custom data processing function (DPF) to the system it is required that you define your own DPF payload structure inside `user_dpf_config.h`. Follow these steps:

1. The payload structure has to consist of `tsk_rte_function_id_t function_id;` as its first field. You can decide the rest of the content. For example, use ids of buffers to refer to them.
2. The custom DPF must be consistent with the DPF interface: `tsk_rte_status_t fn_name(__uncached tsk_rte_function_payload_t* payload)`. Type `tsk_rte_payload_t` is automatically generated and will contain a field with a name matching the type name without suffix `"_t"`.
3. The custom DPF should return a value matching one of `tsk_rte_status_t` elements like `TSK_RTE_OKAY` or `TSK_RTE_ERROR`. Add the function itself to `user_dpf_functions.h`.
4. With the custom payload and DPF use `REGISTER_DPF(DPF_STRUCT, FUNCTION_NAME)` inside `user_dpf_config.h` to add a custom DPF to the system.

2.4.2. Adding More Jobs and Increasing Number of Operations

You can add more job slots in `user_pools_config.h` by using `JOB_POOL_ELEMENT(ITEM_NAME, MAXIMUM_NUMBER_OF_OPERATIONS_FOR_THIS_JOB)`. The provided name will be later referenceable in job claiming.

2.4.3. Adding More Queues and Increasing Their Sizes

You can add more queues in `user_pools_config.h` by issuing `QUEUE_POOL_ELEMENT(ITEM_NAME, MAXIMUM_NUMBER_OF_JOBS_IN_QUEUE, MAXIMUM_NUMBER_OF_OPERATIONS_PER_JOB)`. The provided name will be later referenceable in queue claiming. You can control the amount of jobs that the queue can handle by specifying the `MAXIMUM_NUMBER_OF_JOBS_IN_QUEUE` parameter. Jobs in the queue will host a maximum of `MAXIMUM_NUMBER_OF_OPERATIONS_PER_JOB` operations inside.

Note that a job declared by `JOB_POOL_ELEMENT` can have a larger maximum number of operations than a queue's job can handle. When adding a job to a queue and the number of operations of the job exceeds the maximum number of operations of the jobs of that queue, then an error is returned and the job is not added to the queue.

2.4.4. Adding More Memory Partitions

You can add memory buffers by allocating and registering them in `MEM_POOL_ELEMENT(ITEM_NAME, C_ARRAY_NAME)` in file `user_pools_config.h`. The `ITEM_NAME` is later referenceable in the claim process. The `C_ARRAY_NAME` should be a name of an array that will serve as a buffer.

2.4.5. Setting Up the Interrupts Used by Notification Method Callback

The PPU Run-time uses interrupts for jobs with notification method callback. This is done to make sure that the user specified notification function runs on the same core as was used to submit the job.

Notification is done in two steps. First the PPU signals a TriCore core that a job with notification callback has completed. The `INT_NOTIFY_PPU(CPU, PRIO)` configuration item specifies which TriCore core (TOS=Type Of Service) and what interrupt priority (SRPR=Service Request Priority Number) to use for a `SRC_PPUICI` interrupt. The interrupt handler for `SRC_PPUICI` triggers a second interrupt that calls the notification handler for the job. You can configure the interrupts used for this second interrupt by using `INT_NOTIFY_CPU(CPU, PRIO, GPSR, SR)`, where `CPU` is the TriCore (TOS 0..5), `PRIO` is the priority (SRPN (0..255)), `GPSR` is the General Purpose Service Request Group (GPSRx, for supported values see the processor user guide) and `SR` is the Service Request (0..7). `GPSR` and `SR` define which general interrupt is used (e.g. `SRC_GPSR0SR0`, `SRC_GPSR1SR0`, `SRC_GPSR2SR0`, `SRC_GPSR3SR0`, `SRC_GPSR4SR0` and `SRC_GPSR5SR0`).

2.4.6. RTE Startup Core

The PPU Run-time can be used by multiple TriCore cores. Each core that wants to use the PPU Run-time needs to call the `tsk_rte_init` function. For correct initialization the PPU Run-time has a 'main' core does the initialization of the internal global data structures and the configuration of the interrupt handlers. The other cores will wait in their `tsk_rte_init` until the 'main' core signals the initial initialization is done. The 'main' core is specified using `MAIN_CPU(CPU)` with `CPU 0..5`. When you specify a core different from `CPU0` then you need to make sure that access control is setup such that the CPU is allowed to configure the interrupts used by the notification method callback.

2.5. Using RTE in AUTOSAR

The `CDD_PpuArLayer` is an AUTOSAR module, it functions like most AUTOSAR BSW or MCAL modules. It acts like a layer over the RTE library functionality, providing access to the RTE library interfaces for an AUTOSAR RTE. It has an initialization function which takes care of initializing all the data required for the module to function properly. It has a runnable which takes care of the cyclic events that need to happen for the module to function properly. It provides interfaces for an AUTOSAR SWC to access the RTE library functionality.

Because it is just a layer over the core functionality, any configuration and specifics needed to run the RTE library need to be applied by you as if the RTE library would run in a standalone environment.

2.5.1. Installation/Integration

The main components are:

- RTE library files;

- CDD_PpuArLayer source files;
- CDD_PpuArLayer configuration `.arxml`
- User_config files.

If there is a specific linker and/or compiler configuration that is mandatory for the RTE library files to work, you as an AUTOSAR user should integrate that configuration in your linker/compiler along with whatever configuration is already there in the AUTOSAR environment. There are no linker recommendations and/or specifics for CDD_PpuArLayer sources. Because it is an AUTOSAR CDD, you need to fit the CDD in your AUTOSAR linker environment.

To use the PPU capabilities, you need to configure the memory buffers and other parameters (queue, jobs) as described in the RTE library configuration guide/user manual. You also need to emulate (update/configure) that configuration in the CDD_PpuArLayer using the `CDD_PpuArLayer_Cfg.c` and `.h` along with the configuration data needed by the layer module.

The RTE library files and the CDD_PpuArLayer source files need to be included in the compilation environment of the AUTOSAR project.

The CDD_PpuArLayer configuration `.arxml` files need to be included in the AUTOSAR tooling environment and a new and updated environment needs to be generated to contain the new AUTOSAR port. The `_Main` and `_Init` runnables need to be scheduled on AUTOSAR RTE OS tasks.

The CDD_PpuArLayer provides a client-server type RTE port, making the RTE library functionality available through simple interfaces to the software components running above the AUTOSAR RTE layer.

2.5.2. CDD_PpuArLayer Functionality

The CDD_PpuArLayer provides AUTOSAR compliant access to predefined data processing functions (e.g. matrix multiplication). The module functions like the majority of the AUTOSAR BSW or MCAL modules having an initialization runnable, a main runnable and service like functions provided through the AUTOSAR RTE to any SWC that wants to use them. After having a correct configuration of the RTE library, `CDD_PpuArLayer_Cfg`, AUTOSAR RTE environment, the logic of the CDD layer is as follows:

- The AUTOSAR RTE OS environment initializes the CDD_PpuArLayer;
- The AUTOSAR RTE OS environment starts calling cyclically the main runnable;
- The SWC adds the data that needs to be processed using the RTE port provided interfaces.
- You can choose between predefined operations and custom operations (custom operations are preconfigured in the RTE library and linked in `CDD_PpuArLayer_Cfg.c`).

The configuration to be done for the CDD_PpuArLayer module is to specify how many RTE library tasks it can handle, the memory buffer configuration to match the RTE library buffer configuration and, if it is used, the custom payload. The RTE library task is an item that contains the details of the data processing that needs to be executed by the RTE library. You can configure the custom payload separately in the RTE library and assign it to the `"custom_payload"` variable in `CDD_PpuArLayer_Cfg.c`, along with the buffer configuration to match the custom payload.

Custom operations

- To use the custom operation capability with the AUTOSAR CDD, you should provide your own implementation of the function in the `ppu/custom_dpf.c`.
- You can mention the number of different custom operations that can be used in the `user_config/user_dpf_config.h` by changing the value of the define `AUTOSAR_CUSTOM_OP_POOL_SIZE`.
- You can configure a maximum of 5 different custom operations that are available (for now) and the access from the AUTOSAR API to the custom operation is done using the indexes (0 - 4) as input parameters for the AUTOSAR custom operation.
- The index used in the custom operation also specifies which payload needs to be configured. For example, if you use the index 0 you need to configure the payload `custom_payload_0`.
- The first parameter in the payload is always `function_id` and the remaining parameters can be defined as per your requirement in the file `user_config/user_dpf_config.h`.
- You should configure the parameter `function_id` with one of the values from the enum `tsk_rte_function_id_t` based on the custom operation you want to use. For example, if you want to use the custom operation of the 8th index in the enum `tsk_rte_function_id_t` then you should configure `function_id` as 8U.
- You can use the memory buffers in the custom operations by configuring the memory buffer parameters in the file `CDD_PpuArLayer_Cfg.c` with `ppuArLayer_vccmBuf[x][y]`.
- `x` is the value of the task ID to which you will add the custom operation. It will either be 0 or 1 if the `CDD_PPUARLAYER_TSK_BUFFER_SIZE` is set to 2.
- `y` is the value of the `buffer_id_dst`/`buffer_id_src` used in write & read operation along with the custom operation in the same task job.
- You can also configure other parameters for custom payload in `CDD_PpuArLayer_Cfg.c` by defining the parameter in the file `user_config/user_dpf_config.h`.

For more details, refer to the example of the matrix XOR operation using `custom_payload_1` in `/examples/rte_autosar_demo/ar_matrix_xor_using_custom_DPF`.

2.5.3. RTE AUTOSAR Example

An example on how to use the provided interfaces is present in the `/examples/rte_autosar_demo/ar_env_sim/ar_tests` directory and works like this:

- A successful module initialization had happened.
 - The system simulates an AUTOSAR OS with an `_init` task and 5 ms cyclic task running on core 0.
- The SWC initializes an RTE library task and gets back an ID to be used further.
- Based on the returned ID, you can add operations like mem reads, mem writes, math operations, etc.

- After the operation list is assembled, you can trigger the RTE library task using the same ID.
- The `_Main` of the module takes over and handles triggering of those operations in the RTE library.
- The same `_Main` handles also the status update of the operation execution.
- The SWC using the same ID can check the status of the task and use the result.
 - The SWC is responsible to clear the memory assigned for the RTE library task using the `CDD_PpuArLayer` interface.

The test is built in the standalone `ar_env_sim` application and can be built using either CMake or `amk`. Building it will result in `autosar_sim.elf`. The implemented example test application can be enabled or disabled via a variable from the debugger environment. The tests consist of a custom DPF XOR operation between 2 matrices (100x100) and an addition operation between 2 matrices (100x100). Both are designed to be executed on core `tc0` individually. By default the addition operation between 2 matrices (100x100) application is built.

To build the application with a custom DPF XOR operation between 2 matrices (100x100) replace the files in `examples/rte_autosar_demo/ar_env_sim/ar_tests` and configuration files in `CDD_PpuArLayer` with the files from `examples/rte_autosar_demo/ar_matrix_xor_using_custom_DPF/ar_custom_dpf_app` and `examples/rte_autosar_demo/ar_matrix_xor_using_custom_DPF/ar_customize_dpf_example_cfg` respectively, and then build the executable again. The operation is sent to the PPU for execution and the result is checked against the same operation executed on the TriCore.

2.6. Using the RTE Configuration Tool

You can use the TASKING RTE configuration tool `tsk_rte_config_tool.py` to generate a configuration for the TASKING RTE. See also [Section 2.4, Baremetal Instructions](#).

The tool supports two kinds of configurations `VTC` and `NO_VTC`. See also section "1.4. Multi-Core Support" in the *TASKING SmartCode - TriCore User Guide*.

Usage:

```
python3 tsk_rte_config_tool.py [options]
```

You can specify the following options:

<code>-h, --help</code>	show this help message and exit
<code>--vtc-dir VTC_DIR</code>	Specify TC source directory for code generation (vtc project)
<code>--tc0-dir TC0_DIR</code>	Specify TC0 source directory for code generation
<code>--tc1-dir TC1_DIR</code>	Specify TC1 source directory for code generation
<code>--tc2-dir TC2_DIR</code>	Specify TC2 source directory for code generation
<code>--tc3-dir TC3_DIR</code>	Specify TC3 source directory for code generation
<code>--tc4-dir TC4_DIR</code>	Specify TC4 source directory for code generation
<code>--tc5-dir TC5_DIR</code>	Specify TC5 source directory for code generation
<code>--ppu-dir PPU_DIR</code>	Specify PPU source directory for code generation
<code>--shared-dir SHARED_DIR</code>	

```
Specify shared source directory for code generation
--config-dir CONFIG_DIR
Specify user configuration directory
--generate-default-config [CONFIG_DIR]
Create a baseline for the user configuration in the
specified directory
--generate-default-cdd-config [CONFIG_DIR]
Create a baseline for the CDD configuration in the
specified directory
```

