|  |  |
| --- | --- |
| Issue |  |
| Topic |  |
| Description |  |
|  | |

# RBTPSW\_TPSWCapsule\_Design.docx

# Change History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SWG Rev.** | **DocRev.** | **Description (What/Why)** | **Date** | **Author** |
| **1.284** | **1.1** | **Removed support for TPSW service calls** | **12.08.2010** | **Schäfer** |
|  | **1.2** | **Changed TPSW handling due to migration to RTA OS (AUTOSAR based OS)** | **7.12.2010** | **Zetlmeisl** |
|  | **1.3** | **Changed TPSW framework for BSW modules encapsulation** | **25.05.2012** | **Schäfer / Zetlmeisl / Riess** |
|  | **1.4** | **Added a hint regarding CONST2P parameters** | **07.09.2012** | **Schäfer / Zetlmeisl / Riess** |
|  | **1.5** | **TPSW framework for multicore**  **Preliminary version** | **19.02.2015** | **Riess** |
|  | **1.6** | **Added Configuration switches, Supervisor Mode Restrictions** | **25.04.2018** | **Gittinger** |
|  | **1.7** | **Added GHS linker limitation of the comma operator** | **04.12.2018** | **Zoll** |
|  | **1.8** | **Added manipulators usage caution note**  **Extended timing protection description** | **24.07.2019** | **Gittinger** |
|  | **1.9** | **Added new userinfo for abort caused by interrupt lock while on SC4. Added more info on MPU regions for P1x and U2A. 9 FUNC parameters allowed (not just 6). Minor adjustements.** | **30.11.2020** | **Dias** |
|  | **2.0** | **Added notes regarding restrictions caused by the use of FOTA. Added notes regarding the use of locks in time-protected functions.** | **01.02.2021** | **Dias** |
|  | **2.1** | **Corrected some wrong example code. Mentioned Enhanced Isolation. Other minor corrections.** | **03.11.2021** | **Dias** |
|  | **2.2** | **Update Test Specification** | **08.08.2022** | **Rakel** |

# Scope

This document is relevant for integrators of TPSWs (third party software) into Gen9.3 ECU and for SW Group RBSYS GRs to get technical details about the interaction of such TPSW capsule.

# Abbreviations & Definitions

*see also TCM:*

*RM\HSW\_Archives\HSW\_System9\_Groups\HSWDOC09\_COMMON\Doc\HSW\_Gen09AbbreviationList.xls\HSW\_Gen09AbbreviationList.xls*

*(working version: K:Drive ...HSW\_Doc\Gen09\Documentation\HSW\_Gen09AbbreviationList.xls)*

TPSW: Third party software

MPU: Memory Protection Unit: Special HW on a Microcontroller that allows restriction and monitoring of software accesses to certain address ranges of the memory map.

RBSYS: System software. SW group in HSW that handles startup and OS interaction

OS: Operating system

RE: Runnable entity. Term taken from AutoSAR to describe runtime unit of a software. Like a process in ESP terms.

LCF: Linker Control File. Usually in \prj folder of a project. Define the memory map of a project.

FUNC: function which belongs to an application (trusted or untrusted)

FOTA: firmware over-the-air

# Contents

[RBTPSW\_TPSWCapsule\_Design.docx 1](#_Toc86823246)

[Change History 2](#_Toc86823247)

[Scope 2](#_Toc86823248)

[Abbreviations & Definitions 2](#_Toc86823249)

[Contents 4](#_Toc86823250)

[1 INTRODUCTION 6](#_Toc86823251)

[Overview 6](#_Toc86823252)

[1.1 System view 7](#_Toc86823253)

[1.2 Reference Documents 7](#_Toc86823254)

[1.2.1 Development Documents 7](#_Toc86823255)

[1.2.2 Other Documents 7](#_Toc86823256)

[2 INTERFACE TO THE OUTSIDE 8](#_Toc86823257)

[2.1 TPSW integration into ESP Gen9.3 8](#_Toc86823258)

[2.1.1 Integration concept: TPSW specific Wrapper 8](#_Toc86823259)

[2.1.2 Untrusted/Trusted APPLICATION and FUNC 11](#_Toc86823260)

[2.1.2.1 Declaring and configuring a TPSW application name 12](#_Toc86823261)

[2.1.2.2 Configuration of the MPU table 12](#_Toc86823262)

[2.1.2.3 Configuration of the lock table 13](#_Toc86823263)

[2.1.2.4 Edition of the linker file 14](#_Toc86823264)

[2.1.2.5 Declaration and definition of FUNCs: 16](#_Toc86823265)

[2.1.3 Calling TPSWs 18](#_Toc86823266)

[2.1.3.1 Void-void FUNCs 19](#_Toc86823267)

[2.1.3.2 FUNCs without return value 19](#_Toc86823268)

[2.1.3.3 FUNCs with return value 19](#_Toc86823269)

[2.1.4 Getting the state of a Application/FUNC 19](#_Toc86823270)

[2.1.5 Passing pointers and more complex parameters (Trusted => Untrusted) 22](#_Toc86823271)

[2.1.5.1 Copy of a single pointer 25](#_Toc86823272)

[2.1.5.2 Copy of an array by length 26](#_Toc86823273)

[2.1.5.3 Copy of an array with length in the first element 27](#_Toc86823274)

[2.1.5.4 Copy of a 0 terminated array 27](#_Toc86823275)

[2.1.5.5 Writing custom manipulators 28](#_Toc86823276)

[2.1.6 TPSW calls to other applications 29](#_Toc86823277)

[2.1.6.1 Calls to trusted Bosch software 29](#_Toc86823278)

[2.1.6.2 Alternative way: privileging pieces of code 30](#_Toc86823279)

[2.1.6.3 Locking interrupts 31](#_Toc86823280)

[2.1.6.4 Calls to other untrusted applications 32](#_Toc86823281)

[2.1.7 Timing protection 32](#_Toc86823282)

[2.1.7.1 General concept 32](#_Toc86823283)

[2.1.7.2 Nested calls 34](#_Toc86823284)

[2.1.7.3 Runtime budget calculation 34](#_Toc86823285)

[2.1.7.4 Limitations 36](#_Toc86823286)

[2.1.8 Stack calculation 36](#_Toc86823287)

[2.1.8.1 Function specific stack size 37](#_Toc86823288)

[2.1.8.2 Global stack size 39](#_Toc86823289)

[2.1.9 Recommendation on usage 39](#_Toc86823290)

[2.1.9.1 Header structure 39](#_Toc86823291)

[2.1.10 Examples for integration and MAP file: 42](#_Toc86823292)

[2.1.10.1 Declaration of the TPSW Applications and MPU Config Table 42](#_Toc86823293)

[2.1.10.2 LCF section configuration 45](#_Toc86823294)

[2.1.10.3 Call of a TPSW function (FUNC) 47](#_Toc86823295)

[2.1.11 Configuration switches 47](#_Toc86823296)

[2.1.11.1 RBFS\_TPSWCapsule 47](#_Toc86823297)

[2.1.11.2 RBFS\_TPSWTestsuite 48](#_Toc86823298)

[2.1.11.3 RBFS\_TPSWSVPRestrictions 48](#_Toc86823299)

[3 REQUIREMENTS 49](#_Toc86823300)

[3.1 Functional Requirements 49](#_Toc86823301)

[3.1.1 Requirements within this SW Group 49](#_Toc86823302)

[3.2 Non-Functional Requirements 50](#_Toc86823303)

[3.3 Constraints and Environmental Conditions 50](#_Toc86823304)

[4 KNOWN OPEN ISSUES 50](#_Toc86823305)

[5 TEST SPECIFICATIONS 51](#_Toc86823306)

[6 DESIGN DECISIONS AND FACTORS 56](#_Toc86823307)

[6.1 TPSW capsule 56](#_Toc86823308)

[6.2 Memory protection 56](#_Toc86823309)

[6.3 Timing protection 57](#_Toc86823310)

[7 ALTERNATIVE SOLUTIONS 58](#_Toc86823311)

[8 DETAILED DESIGN DESCRIPTION 59](#_Toc86823312)

[8.1 Design Overview 59](#_Toc86823313)

[8.2 SW group’s internal structure 59](#_Toc86823314)

[8.3 Internal Interfaces 59](#_Toc86823315)

[8.4 Internal Data Structures 59](#_Toc86823316)

[8.5 Internal Functions used 59](#_Toc86823317)

[9 IMPLEMENTATION DETAILS 59](#_Toc86823318)

[9.1.1 MPU setup for Stack area 59](#_Toc86823319)

[9.1.2 Further details on implementation 60](#_Toc86823320)

[10 ANNEXURE 66](#_Toc86823321)

[10.1 Annexure A 66](#_Toc86823322)

[10.2 Annexure B 66](#_Toc86823323)

[11 FURTHER DEVELOPMENT 66](#_Toc86823324)

[11.1 Items Planned 66](#_Toc86823325)

[11.2 Items OnHold 66](#_Toc86823326)

[12 FEATURE AND RESSOURCE MANAGEMENT 66](#_Toc86823327)

[12.1 Resource consumption of missing features 67](#_Toc86823328)

[12.2 Possibilities to reduce resources 67](#_Toc86823329)

# INTRODUCTION

## Overview

RBTPSW Capsules provides services to encapsulate a third party software (TPSW) integrated into Gen9.3 ESP ECUs given task scheme in such a way that memory access of a TPSW are restricted. This is to avoid and detect side effects due to erroneous TPSWs disturbing other SW parts (especially the ESP) by (unintended) manipulating data. In case a TPSW shows such misbehavior, the TPSW is aborted without affecting the rest of the system. More general speaking, the capsule allows to selectively abort a TPSW that causes CPU exceptions during the TPSW execution. These exceptions are not only raised when the SW accesses memory not valid for the TPSW but also when an undefined instruction is encountered in the TPSW.

RBTPSW Capsule feature list:

* Allows a TPSW run in several task and across several cores
* Protects RAM areas of Bosch SW, stack, peripherals and peripheral Rams
* Optionally limits the net runtime execution of a function (active timing protection)
* Optimized for interaction on data interface filled before and evaluated after the TPSW execution
* Allows a flexible use of different address ranges for third party software on µCs with more MPU channels
* Allows encapsulation for BSW modules that access hardware peripherals
* Allows calls of TPSW with function parameters and return values
* Allows function calls of TPSW to trusted code (RB code) and between TPSWs with parameters and return values
* Allow use of pointers and data buffers as parameters between Bosch and TPSW
* Allow TPSW to use interrupt locks and spin locks
* Protects against global pointers corruption. At each switch to trusted mode or ISR/exception entry:
  + The stack pointer is checked for plausibility
  + The global and text pointers are set to a valid value

## System view

TPSW Capsule is services provided by SW group RBSYS due to tight interaction with exception handling and the OS. A separate design document besides the RBSYS design document is provided because of the different intended readers (TPSW Capsule is mainly relevant for TPSW integrators) and the fact the TPSW Capsule is an optional feature in the ESP Gen9.3 via the switch RBFS\_TPSWCapsule\_ON/ RBFS\_TPSWCapsule\_OFF.

## Reference Documents

### Development Documents

***Hint:***

*Link to any further documents that are relevant to this SW group development*

### Other Documents

RBSYS design document attached to RBSYS SW group.

# INTERFACE TO THE OUTSIDE

Important external interfaces of the TPSW Capsule are for static configuration/integration of a TPSW into ESP. Therefore this chapter also comprises the integration guide.

## TPSW integration into ESP Gen9.3

### Integration concept: TPSW specific Wrapper

The interacting between a TPSW in the ESP is assumed to be handled by a wrapper. The wrapper is a piece of TPSW specific Bosch code that calls the TPSW and uses the TPSW framework. In general, an interaction between TPSW and Bosch can occur at any function call signature with the new TPSW framework.

Ideally (runtime efficient) is a wrapper approach that collects data for and from the TPSW, performs data pre and post processing, potential sanity checks of TPSW data and calls the TPSW’s API as void-void at a specific point in time (see also Abbildung 1: Schematic RAM layout for TPSW capsule: Especially use case: pure data capsule and Abbildung 2: Flow of a TPSW call with Bosch wrapper). This is called a pure data capsule and general approach of the old TPSW framework. But because TPSW APIs can have other signature than void-void and might require calls to ESP code during runtime, the TPSW framework is more flexible now.

The TPSW capsule provided with this document gives generic services for TPSW encapsulation for memory protection to be used by such a TPSW specific wrapper code. As the TPSW specific Bosch wrapper is considered to be trusted/high privileged in the terms of the TPSW capsule, it may access all data of the system (just like other Bosch code). With the use of the TPSW capsule the wrapper is responsible to define the point in the program flow where the TPSW is called i.e. the capsule (the memory protection) becomes active and which APIs from TPSW to ESP and between TPSWs are allowed. When the protection is active, only defined memory ranges become accessible, especially write access are restricted to a RAM window. The wrapper has to define the APIs and the memory ranges and the contents of this RAM window (via linker file, see below). So the wrapper is responsible that capsule of the TPSW is complete in three aspects:

1. The memory map for the TPSW is set up correctly. I.e. the TPSW may read all flash and RAM and may not write RAM data outside the capsule legally because the TPSW would be aborted unintended in that case. It may only access (r/w) of peripherals explicitly allowed in the TPSW wrapper configuration.
2. Data within the capsule (especially buffers that might be allocated by the wrapper in the capsule to allow data exchange between the ESP and the TPSW) may not contain critical data in the sense that an accidental manipulation of that data by the TPSW affects the ESP afterwards. Furthermore, data used in trusted ESP code afterwards has to plausibilized after a TPSW call by the user.
3. Allowed function calls between TPSW and ESP (and between TPSWs) have to be configured (defined and declared) correctly in the TPSW framework. Plausibility checks for correct contents of parameters and return values are added where required.

This requires some design step that has to be done with knowledge of the TPSW’s interfaces.

All of the following described API for the TPSW capsule is available via the header RBTPSW\_TPSWCapsule.h

Abbildung 1: Schematic RAM layout for TPSW capsule: Especially use case: pure data capsule

RAM of TPSW (.data, .bss)

Pot. Exchange buffers, allocated by wrapper

ESP RAM

µC Ram layout of project

TPSW linker section

TPSW RAM write access restricted to this window

Trusted TPSW specific Bosch wrapper exchanges data (at least the data to be written by TPSW) before and after TPSW capsule is switched active and TPSW API is called (besides data passed via API parameters)

RAM Start

ESP RAM

Abbildung 2: Flow of a TPSW call with Bosch wrapper

Trusted Bosch Wrapper Code. Pot. prepare data buf + params for TPSW

TPSW FUNC

(API) execution

Trusted Bosch Wrapper Code. Eval. results +pot. copy data to ESP

TPSW\_CALL/TPSW\_START API

Normal Bosch ESP Process in a Task 🡺

Function calls to ESP or other TPSWs

Note that the flash is not restricted because this would cause a lot of integration effort and restrictions to the TPSW without increasing the protection because flash is read-only anyway. Flash restrictions may apply if FOTA is enabled (see section 9.1.2). Also the whole RAM is read only to simplify integration and allow function calls (see section 2.1.2) in the current TPSW framework. So for RAM, only write access has to be considered in configuration of a TPSW. All other memory areas (peripherals memory and registers) are not accessible at all because already a read access can modify contents here. But they can be configured to be accessible for TPSW via the MPU table (see section 2.1.2.2).

The TPSW specific wrapper is responsible that all RAM data required by the TPSW can be accessed by placing desired global variables inside the capsule, at least the data to be written. Typically read and write data in RAM are not sorted out and placed to TPSW memory area together. Note that in case TPSW gets a pointer to data passed, the data pointed to must also be writable by the TPSW. So also in this case a copy of the data for the TPSW placed inside the capsule might be required.

Note that also a stack protection is done. In case of function call parameters, buffers and return values are handled between ESP and TPSW and between TPSWs by the TPSW framework. More complex cases of parameter passing have to be configured explicitly in the framework (see section 2.1.5).

After the FUNC run, the program returns right after this call with full Bosch access rights even if the TPSW was aborted due to any memory violation.

After a successful TPSW execution, data results from the TPSW can be obtained easily by the wrapper because the wrapper has all access rights and can access the TPSW result data buffers. Plausibility checks on the data contents have to be added specifically by the integrator.

Notes:

* In case a TPSW is allowed to use interrupt locks or spin locks, the system interrupts locks are taken. There is no further monitoring than the regular platform monitoring on interrupt locks. I.e. a TPSW can cause the system to stall.
* Although any function call API signatures are allowed between TPSW and ESP code, a TPSW call has a runtime overhead. So alternatives of using pure data capsule with only a few RBTPSW\_CALLs to minimize the switches are preferable.

### Untrusted/Trusted APPLICATION and FUNC

A TPSW software (Trusted or Untrusted) is defined as a TPSW application consisting of one or more FUNC (untrusted or trusted) that are integrated into ESPs task scheme. A FUNC is a function with/without return value which has from 0 to 9 parameters.

Please note that a TPSW application can be multitasking, i.e. run in different tasks or at different positions in a task. Effectively, any FUNC of a TPSW application can be started from any task of any core.

The FUNCs of a TPSW application share a common RAM address range i.e. the same address are considered to be accessible or forbidden by the TPSW capsule. This implies that the FUNC of one application can exchange data between FUNCs (even between different tasks or core) without overhead by just using global variables.

Furthermore it is assumed that the FUNCs of a TPSW application work tightly together. So if one FUNC fails by e.g. a memory protection fault (i.e. it addressed data that is not part of the capsule), the whole application is considered to be erroneous and aborted.

The definition of TPSW application and FUNCs is part of the TPSW specific Bosch wrapper code.

Defining a TPSW application also has implies defining the address range of the TPSW. Therefore definition has to be done consistently in the linker file and in the C Code.

**Defining TPSWs:**

#### Declaring and configuring a TPSW application name

A TPSW application must be declared and configured in the wrapper code. For that the following macros are provided for usage in the header and source file:

* + H file: RBTPSW\_DECLARE\_UNTRUSTED\_APPLICATION(***TPSWAppName***)
  + C file: RBTPSW\_UNTRUSTED\_APPLICATION\_CONFIG(***TPSWAppName***)

Where ***TPSWAppName***is the name of the application. This name is also used in the:

* MPU table (see Configuration of the MPU table)
* Lock table (see Configuration of the lock table)

Please note that there is no limit for the number of declared applications.

#### Configuration of the MPU table

For each untrusted Application, it is necessary to define the MPU table. Each table entry will configure a MPU channel when a FUNC is running. It is possible to configure several entries. At the first call of a FUNC of the application, a plausibility check is done on the whole table.

**Restrictions:** The MPU table supports currently up to 4 entries.

To configure a MPU table, use the following macros in the .C file of the wrapper:

RBTPSW\_DEFINE\_MPUTABLE(***TPSWAppName***) = {

***ENTRY\_1***,

...,

***ENTRY\_n***,

RBTPSW\_MPUCONFIG\_ENDMARKER

};

The last element of the table must be RBTPSW\_MPUCONFIG\_ENDMARKER. Let us define what the possible values are for ***ENTRY\_x*** (also see MPU documentation in the RH850 specification):

* RBTPSW\_MPUCONFIG\_FROMLCF(***TPSWRange\_x***): Where ***TPSWRange\_x*** is the name of the section used in the linker file (see also section 2.1.2.4). In order to define the linker labels, the following macros should be used:

RBTPSW\_DECLARE\_MPUTABLE\_LCFENTRY(***TPSWRange\_1****)*

*...*

RBTPSW\_DECLARE\_MPUTABLE\_LCFENTRY(***TPSWRange\_N****)*

* RBTPSW\_MPUCONFIG\_BYADDRESS(***StartAdr***, ***EndAdr***): Where:
* ***StartAdr***: start address of the region to open. It must be 4-bytes aligned.
* ***EndAdr***: end address of the region to open. It must be 4-bytes aligned. It must be a strictly higher address than ***StartAdr.***

A detailed example can be found in the RBTPSW test suite in the RBTPSW\_MyTPSWCapsule module.

#### Configuration of the lock table

For each untrusted Application, it is necessary to define the lock table. This table tells a specific TPSW application which spin lock types are allowed. It must always end with the RBTPSW\_LOCKCONFIG\_ENDMARKER end marker. There is no limit for the amount of locks in this table. This must be done in the following way:

RBTPSW\_DEFINE\_LOCKTABLE(***TPSWAppName***) = {

***ENTRY\_1***,

...,

***ENTRY\_n***,

RBTPSW\_LOCKCONFIG\_ENDMARKER

};

Currently the only lock available for the ESP Gen9.3 is the common lock. That means that the only valid value for ***ENTRY\_x*** is ***RBTPSW\_LOCKCONFIG\_COMMONLOCK***. In future implementation, it could be possible to add other locks (no-nest locks).

Notes:

* In the single core case, all entries will be handled as local interrupt lock.
* In the multicore case, using the COMMONLOCK means the spinlock will be acquired and that local interrupts will be locked for the core holding the spinlock.
* If the TPSW doesn’t need any spinlock (e.g. only core local locks or no locks at all), it is possible to leave the lock table empty. This will save runtime execution for this specific TPSW. This can be done as follow:

RBTPSW\_DEFINE\_LOCKTABLE(***TPSWAppName***) = {

RBTPSW\_LOCKCONFIG\_ENDMARKER

};

A detailed example can be found in the RBTPSW test suite in the RBTPSW\_MyTPSWCapsule module.

#### Edition of the linker file

The linker control file allows placement of application memory to certain addresses. It is important that the global or local RAM variables of a TPSW application are grouped together. Therefore some entries in the section area of the linker file are required for a TPSW application integration. This TPSW capsule is only given as an example because it depends on the project. Consult the linker documentation of the Green Hills compiler suite for further details.

The intention is to define the RAM area write-accessible for the TPSW application (read access in RAM is always granted) by defining a linker file section that collects and groups together all RAM data that can be used by the TPSW directly and does not need to be protected from misuse by the TPSW. It usually places .data (initialized RAM data) and .bss (uninitialized RAM data) of the TPSW source files.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* MYTPSW1\_LCF \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

RBTPSW\_SECTION\_START(***TPSWRange***)

/\* LCF entry for TPSW .data inside GRAM \*/

.MYTPSW1\_DATA align(4) :

{

***MODULEx***.o (.data)

***MODULEx***.o (.sdata)

***MODULEx***.o (.data)

***MODULEx***.o (.sdata)

...

} > .

/\* LCF entry for TPSW .bss \*/

.MYTPSW1\_BSS align(4) :

{

***MODULEx***.o (.bss)

***MODULEx***.o (.sbss)

***MODULEx***.o (.bss)

***MODULEx***.o (.sbss)

...

} > .

RBTPSW\_SECTION\_END(***TPSWRange***)

* ***TPSWRange***: It must be the same as declared in RBTPSW\_MPUCONFIG\_FROMLCF(***TPSWRange***).
* ***MODULEx***: Here you can list the c or object files of the TPSW. The syntax is slightly different if a lib is used. These lines make sure that all .data and .bss section of the TPSW are part of the capsule and can be accessed by the TPSW. You can list further RAM objects here that have to be accessible by the TPSW without protection but note that all data in this area can potentially manipulated accidentally by the TPSW and is not safe by this.

Including libraries can be done in the following way:

RBTPSW\_SECTION\_START(***TPSWRange***)

.TPSWRSC\_DATA align(4) :

{

lib\_from\_supplier.a (\*(.data))

lib\_from\_supplier.a (\*(.sdata))

} > .

/\* LCF entry for TPSW .bss \*/

.TPSWRSC\_BSS align(4):

{

lib\_from\_supplier.a (\*(.bss))

lib\_from\_supplier.a (\*(.sbss))

} > .

RBTPSW\_SECTION\_END(***TPSWRange***)

More information can be found in the linker documentation under: *C:\MTC10Tools\Compiler\_GHS\[used version]\manuals\build\_v800.pdf*

**Important notes**:

Please take care on following GHS linker limitation. In case you’re using the comma operator for allocating linker sections inside memory regions (*.data align(4) ABS : > GRAM\_BANK\_A****,*** *GRAM\_BANK\_B*) the order of the sections is not guaranteed anymore. This could result in the TPSW usecase in wrong start or end labels used for setting up the MPU. Therefore you’ve to avoid the comma operator in general for TPSW RAM sections.

*Further details are documented here:* [*https://inside-docupedia.bosch.com/confluence/display/BBMCOM/Ordering+sections+in+memory+regions*](https://inside-docupedia.bosch.com/confluence/display/BBMCOM/Ordering+sections+in+memory+regions)

Another detailed example can be found in the RBTPSW test suite. See also the template linker file HWRef\_D5ED\_TPSW.ld\_tpl

#### Declaration and definition of FUNCs:

Functions inside TPSW Applications (FUNCs) must be declared and defined in the wrapper code, using the provided macros shown below:

* H file: RBTPSW\_DECL\_FUNC\_<***Ret***>\_<***n***>PARAM(***RetVal***, ***FuncName***, ***Type\_1***, ***Param\_1***, ..., ***Type\_n***, ***Param\_n***)
* C file: RBTPSW\_DEF\_FUNC\_<***Ret***>\_<***n***>PARAM(***TPSWAppName, Stack, StackO***, ***timelimit,*** ***RetVal***, ***FuncName***, ***Type\_1***, ***Param\_1***, ..., ***Type\_n***, ***Param\_n***)

Where:

* <***Ret***>: if the FUNC has a return value, it must be replaced by ***RET*** else, it must be replaced by ***NORET***
* <***n***>: this is the number of parameter of the FUNC. It can take values from 0 to 9.
* ***RetVal***: is the type of the return value of the FUNC. This parameter is only present if the FUNC has a return value.
* ***FuncName***: this is the name of the FUNC
* ***Timelimit***: If timing protection is enabled: execution time budget of function in microseconds or RBTPSW\_NO\_TIMELIMIT. If timing protection is disabled, this setting is ignored.
* ***Stack***: stack consumption of the actual FUNC
* ***StackO***: stack overhead of the manipulators (for more information see “Passing pointers and more complex parameters”).
* ***Type\_x***: this is the type of the parameter number x. It can be a normal variable or a pointer.
* ***Param\_x***: this is the name of the parameter number x.

Note 1: the TPSW framework internally works on fully resolved C function name (all potential macros resolved). Therefore, on a “#define” pseudo function, no TPSW boundary can be realized. ***FuncName*** is the full resolved C function. It is required for all subsequent definitions (especially manipulators).

Note 2: if the FUNC has parameters, some macros (manipulator) must be defined for passing complex parameters (e.g. pass the content of a pointer) before (pre-manipulator) or after (post-manipulator) the execution of the FUNC. The manipulators can stay empty in case all parameters are normal variable or if no pointers content must be passed. In this case, please set ***StackO*** to 0. For more information, please refer to section 2.1.5.

Concretely, the possible declarations / definitions for a FUNC look like this:

|  |  |
| --- | --- |
| Declaration (H file) | Definition (C file) |
| RBTPSW\_DECL\_FUNC\_NORET\_0PARAM  RBTPSW\_DECL\_FUNC\_NORET\_1PARAM  RBTPSW\_DECL\_FUNC\_NORET\_2PARAM  RBTPSW\_DECL\_FUNC\_NORET\_3PARAM  RBTPSW\_DECL\_FUNC\_NORET\_4PARAM  RBTPSW\_DECL\_FUNC\_NORET\_5PARAM  RBTPSW\_DECL\_FUNC\_NORET\_6PARAM  RBTPSW\_DECL\_FUNC\_NORET\_7PARAM  RBTPSW\_DECL\_FUNC\_NORET\_8PARAM  RBTPSW\_DECL\_FUNC\_NORET\_9PARAM  RBTPSW\_DECL\_FUNC\_RET\_0PARAM  RBTPSW\_DECL\_FUNC\_RET\_1PARAM  RBTPSW\_DECL\_FUNC\_RET\_2PARAM  RBTPSW\_DECL\_FUNC\_RET\_3PARAM  RBTPSW\_DECL\_FUNC\_RET\_4PARAM  RBTPSW\_DECL\_FUNC\_RET\_5PARAM  RBTPSW\_DECL\_FUNC\_RET\_6PARAM  RBTPSW\_DECL\_FUNC\_RET\_7PARAM  RBTPSW\_DECL\_FUNC\_RET\_8PARAM  RBTPSW\_DECL\_FUNC\_RET\_9PARAM | RBTPSW\_DEF\_FUNC\_NORET\_0PARAM  RBTPSW\_DEF\_FUNC\_NORET\_1PARAM  RBTPSW\_DEF\_FUNC\_NORET\_2PARAM  RBTPSW\_DEF\_FUNC\_NORET\_3PARAM  RBTPSW\_DEF\_FUNC\_NORET\_4PARAM  RBTPSW\_DEF\_FUNC\_NORET\_5PARAM  RBTPSW\_DEF\_FUNC\_NORET\_6PARAM RBTPSW\_DEF\_FUNC\_NORET\_7PARAM  RBTPSW\_DEF\_FUNC\_NORET\_8PARAM  RBTPSW\_DEF\_FUNC\_NORET\_9PARAM  RBTPSW\_DEF\_FUNC\_RET\_0PARAM  RBTPSW\_DEF\_FUNC\_RET\_1PARAM  RBTPSW\_DEF\_FUNC\_RET\_2PARAM  RBTPSW\_DEF\_FUNC\_RET\_3PARAM  RBTPSW\_DEF\_FUNC\_RET\_4PARAM  RBTPSW\_DEF\_FUNC\_RET\_5PARAM  RBTPSW\_DEF\_FUNC\_RET\_6PARAM  RBTPSW\_DEF\_FUNC\_RET\_7PARAM  RBTPSW\_DEF\_FUNC\_RET\_8PARAM  RBTPSW\_DEF\_FUNC\_RET\_9PARAM |

Please note that the number of FUNCs within an application is not limited. It is useful to have a look to the examples below.

### Calling TPSWs

The calls which will be described in this section invoke the FUNC ***FUNCName*** (i.e. the function named ***FUNCName***) of the corresponding TPSW application using the memory protection defined for that application. After this point, the SW can only write the restricted RAM memory range configured in the MPU table.

If the FUNC has parameters, they will be passed to the function through a copy in the stack of the corresponding FUNC. This stack region is untrusted and accessible for the FUNC.

Note that in case a TPSW consists of more than one FUNC and the FUNC running at the higher priority task preempted the FUNC running at the lower priority task and the higher priority FUNC causes a fault, not only the higher priority FUNC is aborted immediately but also the lower priority one will be aborted as soon as the task scheme reaches the lower priority FUNC again. So in case a FUNC in a TPSW caused an error, no single instruction of the entire TPSW including all FUNCs of the TPSW is executed anymore making the capsule as tight as possible (this is called “*cascading abort*”).

Similarly, if a FUNC is killed on one core, a “*cross core kill*” mechanism will kill all running FUNC from all other cores by using the “*cascading abort*” mechanism on them.

**Restrictions**:

* In case a TPSW is aborted, the return value and “out parameters” of a FUNC are undefined.
* The “*cross core kill*” mechanism will only kill FUNC which are running in task context. FUNC which are running in CAT2 ISR context will continue their execution.

To call a TPSW from the TPSW specific Bosch wrapper using TPSW capsule you have to use the macro RBTPSW\_CALL or RBTPSW\_START. These macros must never be called under interrupt lock. This would kill the called untrusted application.

#### Void-void FUNCs

In this case, there are two possibilities to call the function:

* **RBTPSW\_CALL**(*FUNCName*());
* AppState = **RBTPSW\_START**(*FUNCName*); RBTPSW\_START returns the state of the corresponding application. This will be described in the next section. This is similar to the old TPSW framework implementation.

#### FUNCs without return value

**RBTPSW\_CALL**(*FUNCName*(param\_1, ..., param\_n));

#### FUNCs with return value

ret = **RBTPSW\_CALL**(*FUNCName*(param\_1, ..., param\_n));

### Getting the state of a Application/FUNC

After calling a FUNC it is important to check if it ran successfully or if it caused an error. There are different possibilities:

* By starting a void-void FUNC with rbtpsw\_appstatus\_t **RBTPSW\_START**(*FUNCName*), the value returned contains the state of the corresponding Application. For example:

AppState = **RBTPSW\_START**(*FUNCName*);

* By calling rbtpsw\_appstatus\_t **RBTPSW\_GetAppStatus**(*TPSWAppName*). The return value contains the state of the Application ***TPSWAppName***.
* By calling rbtpsw\_appstatus\_t **RBTPSW\_GetAppStatusByFunc**(*FUNCName*). The return value contains the state of the Application containing ***FUNCName***.

The return value of these calls can be used to get the status of the TPSW after the run. Its type is rbtpsw\_appstatus\_t and can take the following values:

* **ACTIVE** means that the run was successful. TPSW application is considered to be fault free.
* **NOT\_ACTIVE** means the FUNC and the corresponding application with all other FUNCs were aborted. More information about the type of abort (execution abort or software abort) can be found in the debug structure, as seen below.

As there might further extensions it is best the check the return value for != ACTIVE to get information about an abnormal execution of the TPSW. The Bosch wrapper code has to define and implement the desired TPSW specific reaction on such a fault like logging in EEprom.

**Caution**: in case the status of a TPSW is != ACTIVE, the out parameters and especially return value may not be used (return value can even be out of range). Therefore make sure to avoid incomplete checks such as:

**if(**ret **!=** E\_OK**)** **{** /\* ... User specific reaction here ... \*/ **}**

Instead, the status of the TPSW must be prompted before checking for the return value of the FUNC:

ret **=** RBTPSW\_CALL**(**MyFunc**());**

TPSW\_Status **=** RBTPSW\_GetAppStatus**(**MyApp**);**

**if(**TPSW\_Status **!=** ACTIVE**)**

**{**

/\* ... User specific reaction here ... \*/

**}**

**else**

**{**

**if(**ret **!=** E\_OK**)** **{** /\* ... User specific reaction here ... \*/ **}**

**}**

An exception caused by a TPSW (i.e. status different than ACTIVE) is considered to be a severe fault. Therefore a restart of the TPSW is not possible and not allowed in this ignition cycle. Calling a RBTPSW\_START() or RBTPSW\_CALL() of an aborted TPSW is ignored i.e. no execution of the TPSW code is done.

If a TPSW failed, the TPSW framework fills a debug structure in order to get further details about the failing call. It is possible to read it be calling:

rbtpsw\_debuginfo\_t **RBTPSW\_GetDebugInfo**(*TPSWAppName*)

Note: the debug structure contains only valid data in case the status was different than ACTIVE for a given TPSW application.

The debug structure looks like this:

/\* Debug information for a TPSW application. \*/

**typedef** struct **{**

uint32 userinfo**;** /\* user info when tpsw killed by SW or Enhanced Isolation \*/

uint32 FaultAdr**;** /\* RH850 FEPC register content: PC when exception occurred \*/

rbtpsw\_corefnptr\_t CallCoreFnPtr**;** /\* RBTPSW\_FnCall\_Core\_XXX in which the failure occurred \*/

uint32 rh850\_mei**;**  /\* MEI: Memory error information \*/

uint32 rh850\_mea**;** /\* MEA: Memory error address \*/

uint32 rh850\_feic**;** /\* FEIC(bits 0..15): exception which caused abort: \*/

/\* 0x90/0x91(MIP/MDP), 0xA0(PIE), 0x60(RIE), 0xC0(MAE) \*/

uint32 rh850\_fepsw**;**/\* FEPSW: state of the PSW at error \*/

uint32 rh850\_fpsr**;** /\* FPSR: Floating-point operation setting/status \*/

uint32 rh850\_fpepc**;**/\* FPEPC: Floating-point operation exception program counter \*/

/\* floating point registers are only relevant if the mpu exception was \*/

/\* triggered by an fpu exception. \*/

uint8 os\_status**;** /\* detailed os status \*/

rbtpsw\_debugappstatus\_t debug\_status**;**

**}** rbtpsw\_debuginfo\_t**;**

In case of an abort of the TPSW due to an exception (i.e. addresses out of the TPSW capsule were used or an undefined instruction was encountered), the fault address is filled with debug values.

* The **debug\_status** enumeration:
  + DEBUG\_ACTIVE (0): TPSW is running correctly.
  + DEBUG\_EXC\_ABORTED (1): means the FUNC and the corresponding application with all other FUNCs were aborted due to a severe fault. Forbidden data was accessed or executed (i.e. outside the capsule). Program flow executed outside the allowed range or an undefined instruction was encountered.
  + DEBUG\_SW\_ABORTED (2): means that the Application was aborted by software and not by the MPU. This value can be set if the FUNC call was done with for example: interrupt lock, wrong MPU table configuration … See also the **userinfo** section.
  + DEBUG\_TIMING\_ABORTED (3): the TPSW was killed because its execution net time budget expired.
* The **FaultAdr** pointer contains the program counter (address) which was executed when the exception occurred.
* **CallCoreFnPtr** helps to find out which FUNC caused the failure: it contains the flash address to the function named: RBTPSW\_FnCall\_Core\_***FUNCName***. You will find this function by looking for this address in the MAP file.
* In case the application status (for example by calling RBTPSW\_GetAppStatus(***TPSWAppName***)) returns SW\_ABORTED, the structure element **userinfo** gives more information about the failure:
* 0: no user information is available
* RBTPSW\_USERINFO\_LOCKACTIVE (1): some TPSW API were called/terminated under interrupt lock
* RBTPSW\_USERINFO\_PLAUSIBILITY (2): invalid configuration of the MPU or lock table
* RBTPSW\_USERINFO\_STACKOUTOFRANGE (3): the requested stack range is out of the global stack range, i.e. the system stack dimension is too small (see stack allocation chapter).
* RBTPSW\_USERINFO\_WRONGLOCK (4): attempted to use a lock that is not configured
* RBTPSW\_USERINFO\_LOCKACTIVE\_TRUSTED (5): interrupts locked during TPSW calls from TRUSTED
* RBTPSW\_USERINFO\_LOCK\_WITH\_TP (6): attempted to lock interrupts in a function that has an active time budget – see more information in 2.1.6.3 Locking Interrupts.
* RBTPSW\_USERINFO\_KILLED\_VIA\_EI (9): the TPSW was killed because Enhanced Isolation found an issue. More information about Enhanced Isolation can be found in 6.1 TPSW capsule and OS integration.

The return information of the **RBTPSW\_GetAppStatus()**, **RBTPSW\_GetAppStatusByFunc()** and **RBTPSW\_GetDebugInfo()** API always return application based information, not FUNC based information. So the root cause is stored in the debuginfo[] structure.

### Passing pointers and more complex parameters (Trusted => Untrusted)

As already mentioned in the sections 2.1.2.4 and 2.1.3, it is possible to pass parameters to a TPSW FUNC. When a parameter contains a pointer to the caller RAM region, it is intentionally not possible for an untrusted FUNC to write it. Therefore, we need a technique when calling an untrusted function from a trusted or untrusted function. Note that passing read only buffers and structures via pointer is possible and does not require the manipulators described below.

To solve this problem, the TPSW framework provides manipulators which can redirect the passed pointer to the callee stack and copy the corresponding buffer in it. The integrator of the TPSW can write them to copy the caller RAM region to the callee stack before the actual FUNC call (**pre manipulator**) or/and to copy the buffer from the callee stack to the caller RAM region (**post manipulator**). As the pre manipulator only reads a trusted RAM region, it can be executed as untrusted (the whole RAM is read only in user mode). The post manipulator needs to write in the caller buffer, this is why it is executed as trusted (RAM is only writable in privileged mode).

**Caution**: Only use manipulators for synchronous operations on the buffers during runtime of the untrusted TPSW FUNC. Do not use manipulators if the untrusted TPSW FUNC initiates asynchronous operations using the temporary buffers created by the manipulators on the callee stack. The asynchronous operation would possibly read from/write to the temporary buffers after the TPSW FUNC has returned! When the TPSW FUNC returns the buffers on the stack are no more valid and code using these buffers could corrupt the stack or read invalid contents.

In the following diagram, please consider this:

Purple code block is untrusted

White code block is trusted

Yellow code block belongs to the caller

TPSW\_CALL(MyFunc(p2buffer));

Callee stack

(untrusted)

Caller buffer

Callee buffer

p2buffer

Copy parameters to callee stack

**MyFunc**(p2buffer) execution

- p2buffer now points to the UT stack

- TPSW framework plausibility checks

**Pre Manipulator:**

- Redirect pointer: save p2buffer and p2buffer = p2buffer\_redirected

- Copy caller buffer in callee stack (untrusted)

Execution time

Restoration of caller mode

Caller SW (Bosch or untrusted)

**Pre manipulator**

**Post manipulator**

Caller SW (trusted or untrusted)

**Post manipulator:**

- Copy buffer from callee (UT) stack to the caller buffer back

- Restore p2buffer. It points now to the caller buffer

- Copy return value

p2buffer\_redirected

The pre/post manipulators macros look like this:

**#define** RBTPSW\_PARAM\_MANIP\_PRE\_<***FuncName***> /\* optional pre-manipulator must be implemented here \*/

**#define** RBTPSW\_PARAM\_MANIP\_POST\_<***FuncName***> /\* optional pre-manipulator must be implemented here \*/

Standard manipulators are also provided by the TPSW framework. The integrator can write the pre and post manipulator by itself or use the existing helper manipulators described in the next section.

Note: When using several manipulators for one FUNC, the pre-manipulator has an increasing order, the post-manipulator must have a decreasing order. For example:

#define RBTPSW\_PARAM\_MANIP\_PRE\_***FuncName*** /\* PreManipulator ***1*** \*/ \

/\* PreManipulator ***2*** \*/ \

/\* PreManipulator ***3*** \*/

#define RBTPSW\_PARAM\_MANIP\_POST\_***FuncName*** /\* PostManipulator ***3*** \*/ \

/\* PostManipulator ***2*** \*/ \

/\* PostManipulator ***1*** \*/

#### Copy of a single pointer

This helper is used in the example. Please refer to: 2.1.7

* + **RBTPSW\_PARAM\_IN\_CPYPTR\_PRE(type, pointer)** and **RBTPSW\_PARAM\_IN\_CPYPTR\_POST(type, pointer)**.

If an input pointer of type **type\* pointer** is passed to a FUNC, this helper will copy the content of type **type** from the caller buffer **pointer** to the callee stack, and will redirect **pointer** to it. After this, the FUNC is called.

Note that this manipulator in the in direction is only required if the callee wants to use the buffer as scratch variable.

* + **RBTPSW\_PARAM\_OUT\_CPYPTR\_PRE(type, pointer)** and **RBTPSW\_PARAM\_OUT\_CPYPTR\_POST(type, pointer)**.

If an output pointer of type **type\* pointer** is passed to a FUNC, this helper will redirect **pointer** to the callee stack. After this, the FUNC is called. After the FUNC was called, the content of the callee buffer is copied to the caller buffer and **pointer** is restored.

* + **RBTPSW\_PARAM\_INOUT\_CPYPTR\_PRE(type, pointer)** and **RBTPSW\_PARAM\_INOUT\_CPYPTR\_POST(type, pointer)**.

If an input/output pointer of type **type\* pointer** is passed to a FUNC, this helper will first perform **RBTPSW\_PARAM\_IN\_CPYPTR\_PRE** and then **RBTPSW\_PARAM\_OUT\_CPYPTR\_POST**.

If you use this helper, the stack overhead will be returned by:

RBTPSW\_PARAM\_CPYPTR\_STACKADDER(***type***)

This value should be added to the parameter ***StackO*** at FUNC definition (see section 2.1.2.4).

#### Copy of an array by length

* + **RBTPSW\_PARAM\_IN\_CPYARRAY\_BYLENGTH\_PRE(type, ptr2first, maxlength, length)** and **RBTPSW\_PARAM\_IN\_CPYARRAY\_BYLENGTH\_POST(type, ptr2first, maxlength, length)**.

If an input pointer of type **type\* ptr2first** points to an array, this helper will copy **length** elements of the array to the callee stack before the actual call of the FUNC. The pointer **ptr2first** is redirected to the callee stack before the call of the FUNC. The parameter **length** must be replaced by a FUNC parameter or a constant. The parameter **maxlength** must be a constant.

* + **RBTPSW\_PARAM\_OUT\_CPYARRAY\_BYLENGTH\_PRE(type, ptr2first, maxlength, length)** and **RBTPSW\_PARAM\_OUT\_CPYARRAY\_BYLENGTH\_POST(type, ptr2first, maxlength, length)**.

If an output pointer of type **type\* ptr2first** points to an array, this helper will allocate an array of **maxlength** elements in the callee stack. The pointer **ptr2first** is redirected to this array. After the execution of the FUNC the helper copies **length** elements of the array to the caller buffer. Then **ptr2first** is restored to the caller array. The parameter **length** must be replaced by a FUNC parameter or a constant. The parameter **maxlength** must be a constant.

* + **RBTPSW\_PARAM\_INOUT\_CPYARRAY\_BYLENGTH\_PRE** and **RBTPSW\_PARAM\_INOUT\_CPYARRAY\_BYLENGTH\_POST**.

These helpers are the combination of **RBTPSW\_PARAM\_IN\_CPYARRAY\_BYLENGTH\_PRE** and **RBTPSW\_PARAM\_OUT\_CPYARRAY\_BYLENGTH\_POST**.

If you use this helper, the stack overhead will be returned by:

RBTPSW\_PARAM\_CPYARRAY\_BYLENGTH\_STACKADDER(type, maxlength)

This value should be added to the parameter ***StackO*** at FUNC definition (see section 2.1.2.4).

#### Copy of an array with length in the first element

* + **RBTPSW\_PARAM\_IN\_CPYARRAY\_FIRSTELEM\_PRE(type, ptr2first, maxlength)** and **RBTPSW\_PARAM\_IN\_CPYARRAY\_FIRSTELEM\_POST(type, ptr2first, maxlength)**.

If an input pointer of type **type\* ptr2first** points to an array in caller stack in which the length is in the first element, this helper will copy **ptr2first[0]+1** elements of the array to the callee stack before the actual call of the FUNC. The parameter **maxlength** must be a constant and includes the “length element”[0].

* + **RBTPSW\_PARAM\_OUT\_CPYARRAY\_FIRSTELEM\_PRE(type, ptr2first, maxlength)** and **RBTPSW\_PARAM\_OUT\_CPYARRAY\_FIRSTELEM\_POST(type, ptr2first, maxlength)**.

If an output pointer of type **type\* ptr2first** points to an array, this helper will allocate an array of **maxlength** elements in the callee stack. The pointer **ptr2first** is redirected to this array. After the execution of the FUNC the helper copies **ptr2first[0]+1** elements of the array to the caller buffer. Then **ptr2first** is restored to the caller array. The parameter **maxlength** must be a constant and includes the “length element”[0].

* + **RBTPSW\_PARAM\_INOUT\_CPYARRAY\_FIRSTELEM\_PRE(type, ptr2first, maxlength)** and **RBTPSW\_PARAM\_INOUT\_CPYARRAY\_FIRSTELEM\_POST(type, ptr2first, maxlength)**.

These helpers are the combination of **RBTPSW\_PARAM\_IN\_CPYARRAY\_FIRSTELEM\_PRE** and **RBTPSW\_PARAM\_OUT\_CPYARRAY\_FIRSTELEM\_POST**.

If you use this helper, the stack overhead will be returned by:

RBTPSW\_PARAM\_CPYARRAY\_FIRSTELEM\_STACKADDER(type, maxlength)

This value should be added to the parameter ***StackO*** at FUNC definition (see section 2.1.2.4).

#### Copy of a 0 terminated array

* + **RBTPSW\_PARAM\_IN\_CPYARRAY\_0TERM\_PRE(type, ptr2first, maxlength)**and **RBTPSW\_PARAM\_IN\_CPYARRAY\_0TERM\_POST(type, ptr2first, maxlength)**.

If an input pointer of type **type\* ptr2first** points to a 0 terminated array, this helper will copy the n elements of the array to the callee stack before the actual call of the FUNC. The pointer **ptr2first** is redirected to the callee stack before the call of the FUNC. The parameter **maxlength** must be a constant. In case no zero is found, all maxlength elements are copied.

* + **RBTPSW\_PARAM\_OUT\_CPYARRAY\_0TERM\_PRE(type, ptr2first, maxlength)**and **RBTPSW\_PARAM\_OUT\_CPYARRAY\_0TERM\_POST(type, ptr2first, maxlength)**.

If an output pointer of type **type\* ptr2first** points to a 0 terminated array, this helper will allocate an array of **maxlength** elements. The pointer **ptr2first** is redirected to this array. After the execution of the FUNC the helper copies **n** elements of the array to the caller buffer. Then **ptr2first** is restored to the caller array. The parameter **maxlength** must be a constant. In case no zero is found, all maxlength elements are copied.

* + **RBTPSW\_PARAM\_INOUT\_CPYARRAY\_0TERM\_PRE(type, ptr2first, maxlength)**and **RBTPSW\_PARAM\_INOUT\_CPYARRAY\_0TERM\_POST(type, ptr2first, maxlength)**.

These helpers are the combination of **RBTPSW\_PARAM\_IN\_CPYARRAY\_0TERM\_PRE** and **RBTPSW\_PARAM\_OUT\_CPYARRAY\_0TERM\_POST**.

If you use this helper, the stack overhead will be returned by:

RBTPSW\_PARAM\_CPYARRAY\_0TERM\_STACKADDER(type, maxlength)

This value should be added to the parameter ***StackO*** at FUNC definition (see section 2.1.2.4).

#### Writing custom manipulators

The integrator can write custom manipulators: it must follow the following rule:

* The pre-manipulator must always begin with “**{**“ and the post-manipulator must always end with “**}**”

**Notes**:

* It is the responsibility of the integrator to **take into account the stack overhead of the manipulator.** This value should be added to the parameter ***StackO*** at FUNC definition (see section 2.1.2.4).
* For any of these helpers, if the provided length is greater than the max length, the actual copy will be performed on the max length.
* The validity of the out pointers should be checked. This is the responsibility of the TPSW integrator.
* For more examples, please have a look to the TPSW test suite.
* CONST2P parameters in manipulators do not work because of the shadow copy on the stack written by the manipulator. You can fix the problem by:
  + Changing the function signature to a normal pointer.

OR

* + Use a custom manipulator which uses a shadow local variable which is not constant. Warning: this generates a warning at compilation time. For example:

**#define** RBTPSW\_PARAM\_OUT\_CPYCONSTARRAY\_BYSIZE\_PRE(type, ptr2first, sizeofBuf) RBTPSW\_PARAM\_OUT\_CPYCONSTARRAY\_BYSIZE\_PRE\_intern(type, ptr2first, sizeofBuf)

**#define** RBTPSW\_PARAM\_OUT\_CPYCONSTARRAY\_BYSIZE\_PRE\_intern(type, ptr2first, sizeofBuf)\

{\

type \* ptr2first**##**\_cpy\_ptr;\

type ptr2first**##**\_data[sizeofBuf];\

ptr2first**##**\_cpy\_ptr = ptr2first;\

{\

type \* ptr2first = ptr2first**##**\_data;

**#define** RBTPSW\_PARAM\_OUT\_CPYCONSTARRAY\_BYSIZE\_POST(type, ptr2first, sizeofBuf) RBTPSW\_PARAM\_OUT\_CPYCONSTARRAY\_BYSIZE\_POST\_intern(type, ptr2first, sizeofBuf)

**#define** RBTPSW\_PARAM\_OUT\_CPYCONSTARRAY\_BYSIZE\_POST\_intern(type, ptr2first, sizeofBuf)\

}\

(**void**)memcpy((**void**\*)ptr2first**##**\_cpy\_ptr, (**const** **void** \*)ptr2first**##**\_data, sizeofBuf);\

}

### TPSW calls to other applications

#### Calls to trusted Bosch software

When an untrusted SW is running, the TPSW framework provides a solution to call a trusted function (UT => T). Exactly the same mechanism as a T => UT call is used:

* H file: RBTPSW\_DECL\_FUNC\_<***Ret***>\_<***n***>PARAM(***RetVal***, ***FuncName***, ***Type\_1***, ***Param\_1***, ..., ***Type\_n***, ***Param\_n***)
* C file: RBTPSW\_DEF\_FUNC\_<***Ret***>\_<***n***>PARAM(RBTPSW\_STD\_TRUSTED\_APP, 0, 0, RBTPSW\_NO\_TIMELIMIT, ***RetVal***, ***FuncName***, ***Type\_1***, ***Param\_1***, ..., ***Type\_n***, ***Param\_n***)

Note that it is not required to give the stack/stack overhead need of the trusted FUNC. If parameters are required, please also define the manipulators. In this case, they must stay empty.

The call of the trusted FUNC occurs as usual with RBTPSW\_START or RBTPSW\_CALL. These macros must never be called under interrupt lock. This would kill the calling untrusted application. Before starting the trusted FUNC, a stack pointer plausibility check is performed. If it is wrong, the calling application will be killed.

Note 1: In case a FUNC with return value is killed, the returned value is cleared with memset(), i.e. a scalar return value is 0.

Note 2: RBTPSW\_CALL and RBTPSW\_START can be used in the same way to call an untrusted FUNC from another untrusted Application.

The TPSW framework already provides a trusted application: **RBTPSW\_STD\_TRUSTED\_APP**. Please note that only one trusted application is needed. However, it is possible to define other trusted application by using:

* + H file: RBTPSW\_DECLARE\_TRUSTED\_APPLICATION(***TrustedTPSWAppName***)
  + C file: RBTPSW\_TRUSTED\_APPLICATION\_CONFIG(***TrustedTPSWAppName***)

As the application is trusted, it is also not necessary to configure the linker file. We assume that a trusted application won’t cause an error, that’s why it should never be killed. If some error occurs during the call of a trusted function, the corresponding caller untrusted application will be killed and the trusted function call will not be completed.

The following APIs will always return “**ACTIVE**” for a trusted application:

* **RBTPSW\_START**(*FUNCName*)
* **RBTPSW\_GetAppStatus**(*TPSWAppName*)
* **RBTPSW\_GetAppStatusByFunc**(*FUNCName*)

#### Alternative way: privileging pieces of code

It’s possible to privilege a piece of code e.g. let the untrusted TPSW call an ESP function directly (untrust) and make the relevant part inside the function trusted.

* rbtpsw\_CPUmode\_t **RBTPSW\_Switch2PrivMode**(**void**): It returns the current system mode and sets the system mode to “privileged”. Before changing the system mode, this API performs a reset of the global registers, a check of the stack pointer and active locks. If anything is wrong, the current TPSW will be killed.
* **void** **RBTPSW\_RestoreMode**(rbtpsw\_CPUmode\_t PreviousMode) : this function restores the system mode to the last mode used. The parameter “PreviousMode” must be the value returned by RBTPSW\_Switch2PrivMode.

**For example:**

/\* this is a Bosch service code called directly by a TPSW ... \*/

{

tpsw\_CPUmode\_t PrevMode;

PrevMode = **RBTPSW\_Switch2PrivMode**();

/\* ... execute here your trusted code ... \*/

**RBTPSW\_RestoreMode**(PrevMode);

}

Important note: the integrator has the responsibility to call **RBTPSW\_Switch2PrivMode()** and **RBTPSW\_RestoreMode()** at the right place. It is also important to check that the caller is authorized to perform a certain call and that the potential pointers and parameters have a plausible value.

**RBTPSW\_Switch2PrivMode()** and **RBTPSW\_RestoreMode()** also work when it is called in privileged mode. So the code example above could be used for common services which are used by a TPSW or Bosch code.

#### Locking interrupts

As far as possible, interrupts lock or spin lock must be avoided in an untrusted FUNC (e.g. influences the system dynamic behavior).

However, it can be allowed if the untrusted software is a BSW module. This can be done by using the following APIs:

* Interrupt lock:

**void** **RBTPSW\_EnterCorelocalIntlock**(**void**)

**void** **RBTPSW\_ExitCorelocalIntlock**(**void**)

* Spin lock (also locks interrupts in the local core):

**void** **RBTPSW\_EnterLock**(**lockstr\_t\* lock**)

**void** **RBTPSW\_ExitLock**(**lockstr\_t\* lock**)

The “lock” which is taken must be entered in the TPSW specific lock table, see also section 2.1.2.3.

Restrictions:

* If it is needed to lock the interrupts in a FUNC, the above APIs must be called.
* It is not allowed to end a FUNC with locked interrupts or spin lock.
* While the interrupts are locked, it is not allowed to call RBTPSW\_START, RBTPSW\_CALL and RBTPSW\_Switch2PrivMode.
* Releasing a lock cannot be done if it was not taken first. This will kill the current TPSW with SW abort.
* In case the FUNC calling the above APIs is time-protected, the lock will not be taken and there will be a software abort. When there is a time-protected FUNC, which calls a non-time-protected FUNC, the callee is still allowed to acquire the lock. This is because a suppression flag is set and the timer interrupt will not arrive during the callee execution, even if the timer expires. Lock acquisition is safe and allowed in this scenario.

#### Calls to other untrusted applications

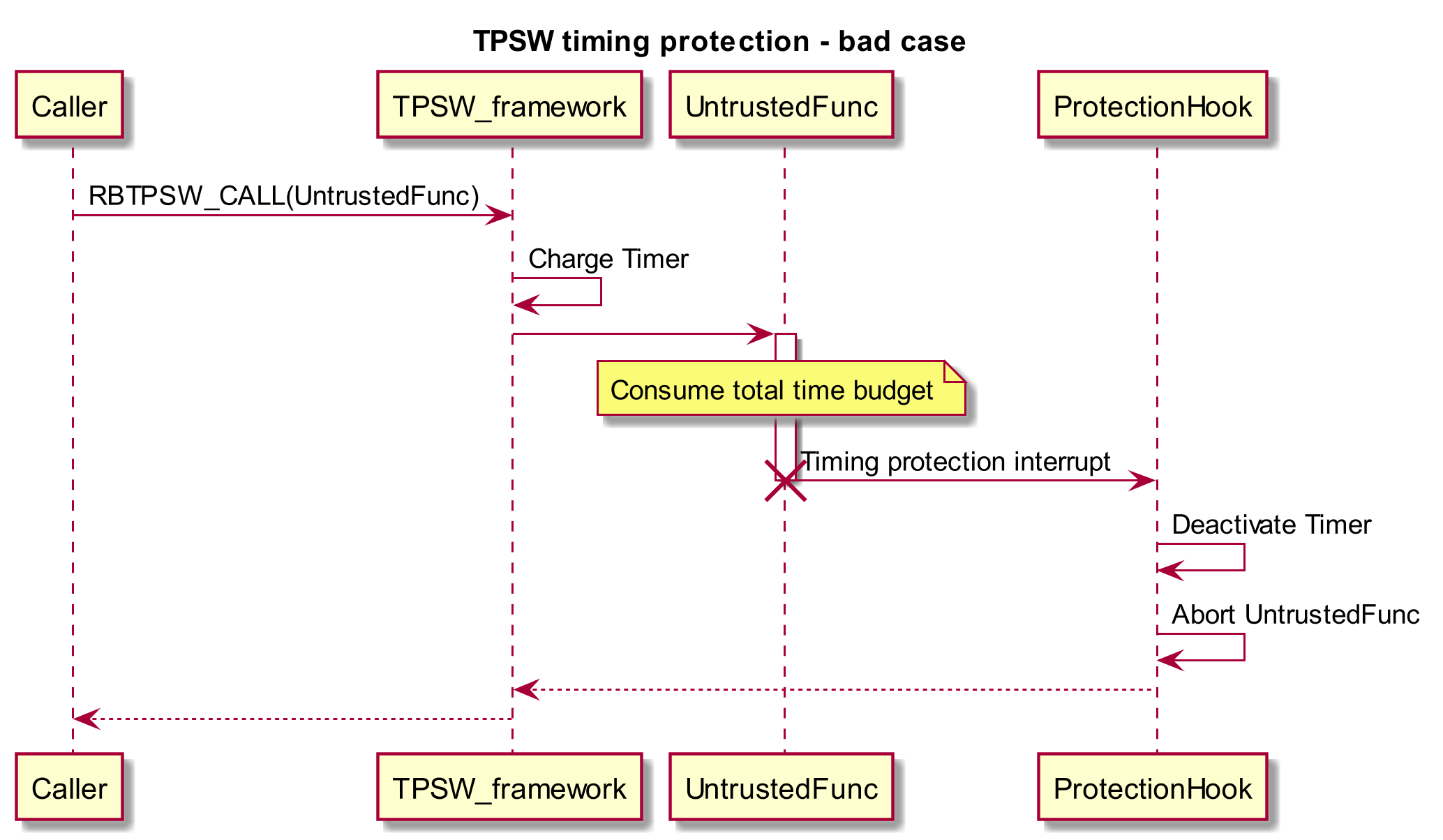
In the same way than for “trusted => untrusted” or “untrusted => trusted” calls, it is also possible to perform “untrusted => untrusted” calls with RBTPSW\_CALL and RBTPSW\_START.

### Timing protection

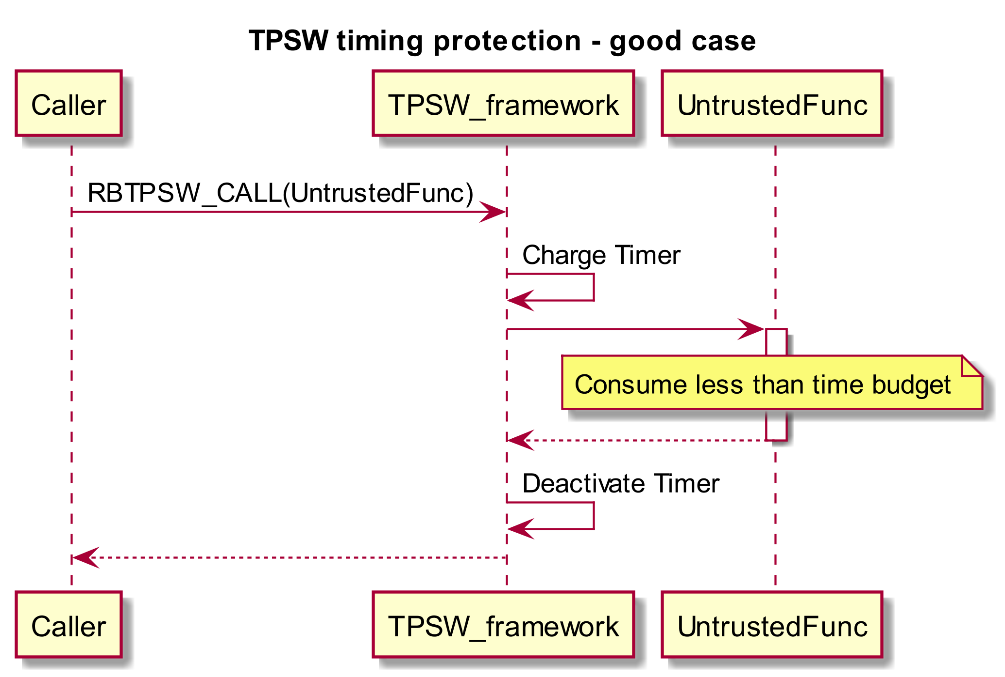
#### General concept

TPSW framework implements active timing protection for untrusted functions using functionality provided by the AUTOSAR OS. TPSW Timing protection can be turned on by configuration switch (see RBFS\_TPSWCapsule\_SC4, section 2.1.11). Timing protection adds an individually configurable execution time budget to each TPSW function at configuration time.

At runtime, before executing a timing protected TPSW function, a timer is charged with the time budget value of the TPSW function to be called. If a timing protection budget has expired a timer interrupt is generated and the TPSW framework aborts the TPSW application. Following sequence chart illustrates this behaviour:



If the TPSW FUNC returns within the timing budget, the timer is uncharged at return time of the TPSW FUNC. This behaviour is illustrated by the following chart:



For checking if the TPSW FUNC has been aborted the same mechanism as described in section 2.1.4 must be used.

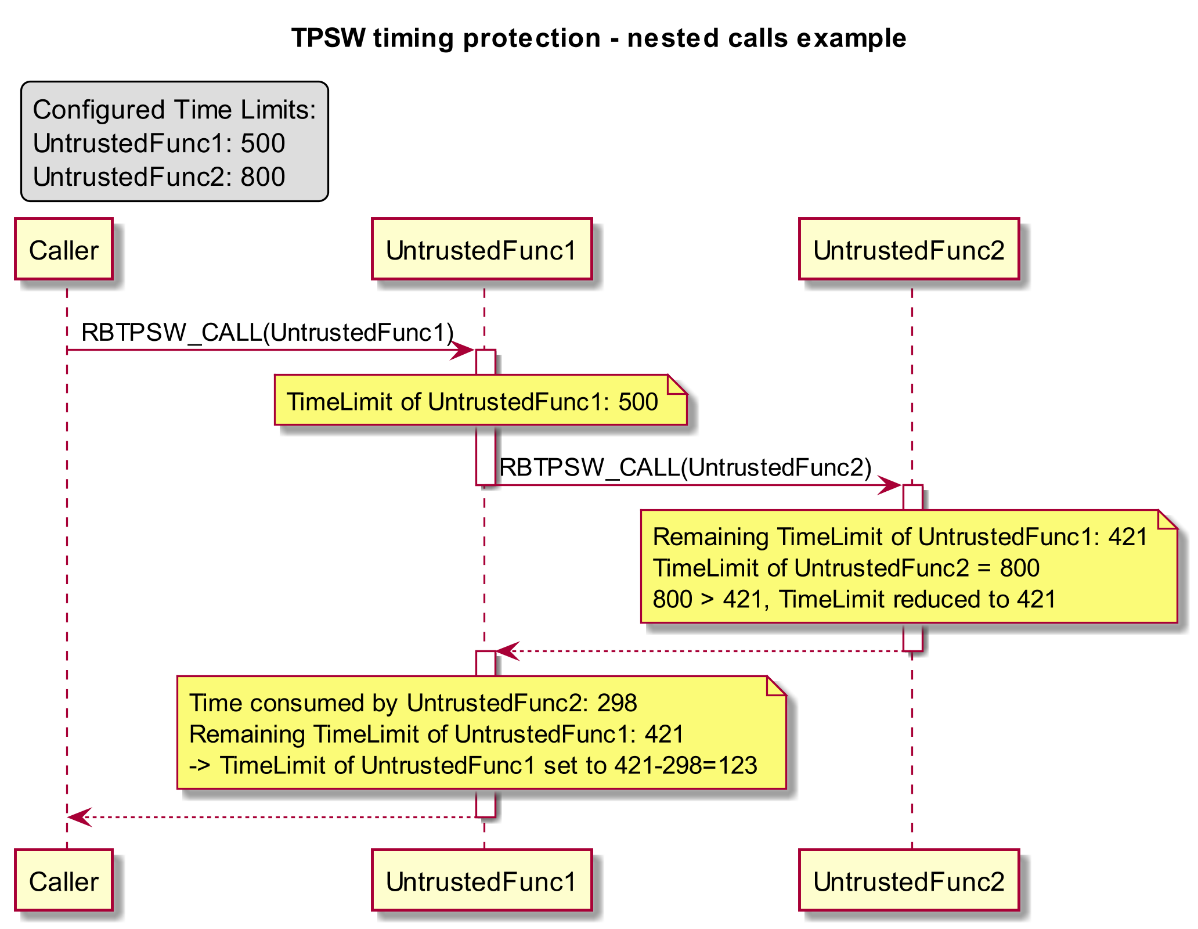
#### Nested calls

Timing protected untrusted TPSW functions can call non timing protected TPSW functions (trusted or not trusted) and vice versa. Timing protected untrusted TPSW functions can also call Timing protected untrusted TPSW functions. The time consumed by the inner function is also charged from the time of the outer function. See details in section 2.1.7.3.

#### Runtime budget calculation

The total call duration made by a timing protected TPSW function including possible nested calls to TPSW functions is charged from its configured time budget:

* If a nested TPSW call ends or was aborted, the time budget of the caller gets reduced by the time consumed by the callee.
* If a nested TPSW call is also a timing protected function and its budget is larger than the remaining budget of the caller, its budget gets limited to the remaining budget of the caller. See example sequence chart below.
* If a nested TPSW call expires both the inner and outer function time budget, the callee and the caller will be killed.
* If the callee is not timing protected or if it is a trusted function or code section delimited by RBTPSW\_Switch2PrivMode()/RBTPSW\_RestoreMode(), the callee can run indefinitely. If the budget of the caller expires during the callee runs, the timing protection interrupt will be delayed until the end of the callee. This is implemented by the suppression flag mechanism (see also: design chapter).



Exception: If a timing protected function gets preempted by a task with higher priority the remaining time budget of a TPSW FUNC is preserved by the OS. The timer is deactivated during execution of the higher priority task. Once the timing protected gets rescheduled again, the timer is reactivated with the remaining time budget.

The time budget of an untrusted function should be dimensioned in a way that the function will not influence the system if it erroneously runs too long. This budget includes:

* The runtime overhead of the Os and of the TPSW framework itself.
* The runtime overhead for preemptions
* The runtime or budget of all the callees (trusted/untrusted and w/o timing protection)

#### Limitations

* Usage of timing protection in a FUNC requires **avoiding interrupt-based locks** in TPSW FUNCs. Background is that the timer interrupt used by timing protection is a CAT 2 interrupt on the same priority level as the RTA-OS. If an interrupt lock prevents RTA-OS from scheduling, the timer interrupt is blocked as well. A TPSW FUNC stalling with an active interrupt lock could not get terminated by the timer interrupt if interrupts are locked.
* The configured time budget of a TPSW FUNC is a **brutto time budget**. The actual netto timing budget of the called TPSW FUNC is some microseconds smaller than the given brutto timing budget by which the timing protection timer is being charged. See details in section 2.1.7.3.

### Stack calculation

In RTA OS there is a single stack policy. So there is no need to place the TPSW stack explicitly in the RAM area of the TPSW. This strategy reduces the overall stack consumption because the global task scheme (other processes, cooperative/preemptive scheduling) can be considered when dimensioning the global stack with respect to TPSW applications.

The TPSW framework dynamically manages the stack handling for each TPSW function. When invoking an untrusted function, it grants access only to the configured size (given in RBTPSW\_DEF\_FUNC\_<*Ret*>\_<*n*>PARAM(*TPSWAppName,* ***Stack*, *StackO***, timelimit, *RetVal*, *FuncName*, *Type\_1*, *Param\_1*, ..., *Type\_n*, *Param\_n*) macro). Any violations will result in killing the TPSW application. However, the framework extension which allows arbitrary function calls between trusted and untrusted applications requires some more aspects to be considered because parameters and return values are now placed and even duplicated onto the stack (partially in the caller and callee area).

In general it must be distinguished between the stack size to be provided for each TPSW untrusted FUNC (RBTPSW\_DEF\_FUNC\_NO/RET\_xPARAM) and the overall global stack size. The main difference is that some parts of the framework run with privileged access rights which only affect the global stack (e.g. OS callbacks).

Therefore, this chapter is split in two parts:

* function specific stack size needed to set-up the MPU for untrusted FUNC
* global stack size

#### Function specific stack size

This section gives some details about the general handling of the function specific stack sizes which is used to set-up the MPU and to be provided via stack and stacko parameter for untrusted FUNCs. The general statement is that most of the framework required stack values are already considered internally. Only a few topics have to be handled when specifying a TPSW.

**Rules for function specific stack sizes**

**If an untrusted function A calls another function B of a different application (trusted or untrusted), the required stack size configured in** RBTPSW\_DEF\_FUNC\_<*Ret*>\_<*n*>PARAM(*TPSWAppName, RetVal*, *FuncName*, timelimit, ***Stack*,** *StackO*, *Type\_1*, *Param\_1*, ..., *Type\_n*, *Param\_n*) **of the caller function A has to be increased by 100 bytes (according (b), see below).**

Examples:

* void MYTPSW1\_FuncA(uint8\_t\* p1) needs 400 bytes and calls MYTPSW2\_FuncB() and a trusted function RB\_xx()sequentially

=> the required stack for FuncA is 400 bytes + 100 bytes for the call to MYTPSW2/RB\_xx + overhead for copy-pointer manipulators

* uint32\_t MYTPSW2\_FuncB(void) needs 600 bytes

=> the required stack for FuncB is just 600 bytes, no overhead for manipulators, no overhead because no trustlevel switch

#define STACKADDER\_FuncA **1\*RBTPSW\_PARAM\_CPYPTR\_STACKADDER(uint8\_t)**

RBTPSW\_DEF\_FUNC\_NORET\_1PARAM(MYTPSW1, (**400+100)**,0, RBTPSW\_NO\_TIMELIMIT, STACKADDER\_FuncA, MYTPSW1\_FuncA, uint8\_t\*, p1)

RBTPSW\_DEF\_FUNC\_RET\_0PARAM(MYTPSW2, **600**, 0, RBTPSW\_NO\_TIMELIMIT, uint32\_t, MYTPSW2\_FuncB)

Details

The following flow shows that both, the caller and callee stack sizes, have to be adapted when doing a function call between applications:

=> stack consumption of caller function **(a)**

=> TPSW framework overhead (on caller stack) **(b)**

=> TPSW framework overhead (on callee stack) **(c)**

=> stack consumption of callee function **(d)**

**(a)** and **(d)** are given by the TPSW supplier and/or determined with your stack analysis tool, such as GHS gstack.

**(b)** is mainly defined by the caller wrapper of the TPSW framework (RBTPSW\_Caller\_##fnname) and OS/MPU set-up overhead.

OS and MPU set-up are not relevant for the untrusted function specific stack because they are running in privileged mode. The required stack for RBTPSW\_Caller\_##fnname consists of a limited number of framework structures which is < 100 bytes. This overhead has to be considered explicitly when an untrusted function calls another function. I.e. the stack size of the caller has to be increased. A call from trusted can be ignored here because the used stack frame is accounted to the global (privileged) stack (off course to be considered there - see next section).

**(c)** is mainly defined by the local buffers for parameters and return values which are allocated additionally on the callee stack (i.e. TPSW internal wrapper function: RBTPSW\_FnCall\_Core\_##fnname) and the manipulator functions. All parameters and return values are already considered by the framework for the dynamic MPU set-up. The stack consumption of the used manipulators has to be specified separately. Macros are provided for the default manipulators. E.g. RBTPSW\_PARAM\_CPYARRAY\_BYLENGTH\_STACKADDER

Example: a function which needs 300 bytes and uses three copy-pointer manipulators has to be defined as:

#define RBTPSW\_STACKADDER\_FUN\_MyA\_UL **3\*RBTPSW\_PARAM\_CPYPTR\_STACKADDER(uint8\_t)**

RBTPSW\_DEF\_FUNC\_RET\_3PARAM(MYTPSW1, **300**,0, RBTPSW\_NO\_TIMELIMIT, RBTPSW\_STACKADDER\_FUN\_MyA\_UL, int32\_t, FUN\_MyA\_UL, uint8\_t\*, p\_in, uint8\_t\*, p\_out, uint8\_t\*, p\_inout)

FUN\_MyA\_UL(uint8\_t\* p1, uint8\_t\* p2, uint8\_t\* p3);

Note: the stack consumed by the Os (CallTrustedFunction() API) is also taken in account: 20 bytes stack are required for each function.

#### Global stack size

To get the global stack overhead, three steps have to be done:

1. Getting the complete TPSW call graph – independent of trusted or untrusted level. Including the manipulator code, framework and OS overhead etc.
2. Adding some offset to compensate the MPU granularity restrictions
3. Considering the nesting level (i.e. task scheme) to get the total stack adder

**Currently no tooling is available for getting the complete call graph and the stack consumption of each function.**

### Recommendation on usage

#### Header structure

It is recommended for a given untrusted/trusted application to gather the Application configuration/declaration and FUNC declaration/definition in a wrapper (\*.c and \*.h file). The header of the wrapper must include all header files which contains the functions which will be used by the TPSW framework. This is required in order to check at compilation time that the definition of the encapsulated function and the definition of the FUNC definition are identical.

* **This is an example of the call of a trusted => untrusted software call.**

**TPSW.c**

Myfunc()

Myfunc2()

**TPSW.h**

Myfunc()

Myfunc2()

**RB.c**

TPSW\_CALL(Myfunc());

TPSW\_CALL(Myfunc2());

**Wrapper.h**

TPSW\_DECL

**Wrapper.c**

TPSW\_DEF

**RBTPSW\_TPSWCapsule.h**

* **The following schematic represents an untrusted => trusted call:**

**RB.c**

Myfunc()

Myfunc2()

**RB.h**

Myfunc()

Myfunc2()

**TPSW.c**

TPSW\_CALL(Myfunc());

TPSW\_CALL(Myfunc2());

**Wrapper.h**

TPSW\_DECL

**Wrapper.c**

TPSW\_DEF

**RBTPSW\_TPSWCapsule.h**

* **The following file structure is recommended when using RBTPSW\_Switch2PrivMode() and RBTPSW\_RestoreMode()**

**TPSW.c**

MyUntrustedFunc()

{

…

**MyTrustedFunc();**

…

}

**TPSW.h**

MyUntrustedFunc()

**RB.c**

TPSW\_CALL(MyUntrustedFunc());

**MyTrustedFunc()**

**{**

**TPSW\_Switch2PrivMode();**

**...**

**TPSW\_RestoreMode();**

**}**

**Wrapper.h**

TPSW\_DECL

**Wrapper.c**

TPSW\_DEF

**RBTPSW\_TPSWCapsule.h**

### Examples for integration and MAP file:

Here is an example of a TPSW Capsule configuration with two applications, using 2 FUNC each.

#### Declaration of the TPSW Applications and MPU Config Table

In this example, 2 untrusted applications are used. Both of them export 2 functions (with/without return value, with/without parameter). The corresponding RAM windows are configured via LCF symbols.

The MYTPSW1 untrusted application contains a void-void function and a function with 2 parameters and a return value. The second parameter is INOUT and is passed through a manipulator.

The MYTPSW2 untrusted application contains a void-void function and a function with 1 parameter and no return value.

**First the header file of the wrapper will be described (wrapper.h):**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MYTPSW1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "mytpsw1.h"

/\* MYTPSW1 declares the following functions \*/

/\* void TPSW1\_Prc(void); \*/

/\* uint32\_t TPSW1\_Mytest\_UL(uint8\_t variable, structure\_t \* structure); \*/

RBTPSW\_DECLARE\_UNTRUSTED\_APPLICATION(MYTPSW1)

RBTPSW\_DECL\_FUNC\_NORET\_0PARAM(TPSW1\_Prc)

RBTPSW\_DECL\_FUNC\_RET\_2PARAM(uint32\_t, TPSW1\_Mytest\_UL, uint8\_t, variable, structure\_t \*, structure)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MYTPSW2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "mytpsw2.h"

/\* MYTPSW2 declares the following functions \*/

/\* void TPSW2\_Prc(void); \*/

/\* void TPSW2\_Mytest\_V(uint8\_t variable1); \*/

RBTPSW\_DECLARE\_UNTRUSTED\_APPLICATION(MYTPSW2)

RBTPSW\_DECL\_FUNC\_NORET\_0PARAM(TPSW2\_Prc)

RBTPSW\_DECL\_FUNC\_NORET\_1PARAM(TPSW2\_Mytest\_V, uint8\_t, variable1)

**The c file looks like this (wrapper.c):**

#include "wrapper.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MYTPSW1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

RBTPSW\_UNTRUSTED\_APPLICATION\_CONFIG(MYTPSW1)

RBTPSW\_DECLARE\_MPUTABLE\_LCFENTRY(MYTPSW1\_LCF)

RBTPSW\_DEFINE\_MPUTABLE(MYTPSW1) = {

RBTPSW\_MPUCONFIG\_FROMLCF(MYTPSW1\_LCF),

RBTPSW\_MPUCONFIG\_BYADDRESS(0xFFF82000UL, 0xFFF83000UL), /\* open a MPU channel for a device register. \*/

RBTPSW\_MPUCONFIG\_ENDMARKER

};

/\* TPSW1\_Prc requires 1000 byte stack \*/

RBTPSW\_DEF\_FUNC\_NORET\_0PARAM(MYTPSW1, 1000, 0, RBTPSW\_NO\_TIMELIMIT, TPSW1\_Prc)

#define RBTPSW\_PARAM\_MANIP\_PRE\_TPSW1\_Mytest\_UL RBTPSW\_PARAM\_INOUT\_CPYPTR\_PRE(structure\_t, structure)

#define RBTPSW\_PARAM\_MANIP\_POST\_TPSW1\_Mytest\_UL RBTPSW\_PARAM\_INOUT\_CPYPTR\_POST(structure\_t, structure)

RBTPSW\_DEF\_FUNC\_RET\_2PARAM(MYTPSW1, 100,0, RBTPSW\_NO\_TIMELIMIT, RBTPSW\_PARAM\_CPYPTR\_STACKADDER(structure\_t), uint32\_t, TPSW1\_Mytest\_UL, uint8\_t, variable, structure\_t \*, structure)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MYTPSW2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

RBTPSW\_UNTRUSTED\_APPLICATION\_CONFIG(MYTPSW2)

RBTPSW\_DECLARE\_MPUTABLE\_LCFENTRY(MYTPSW2\_LCF)

RBTPSW\_DEFINE\_MPUTABLE(MYTPSW2) = {

RBTPSW\_MPUCONFIG\_FROMLCF(MYTPSW2\_LCF),

RBTPSW\_MPUCONFIG\_ENDMARKER

};

RBTPSW\_DEF\_FUNC\_NORET\_0PARAM(MYTPSW2, 2000, 0, RBTPSW\_NO\_TIMELIMIT, TPSW2\_Prc)

#define RBTPSW\_PARAM\_MANIP\_PRE\_TPSW2\_Mytest\_V /\* No pointer to copy: nothing to do \*/

#define RBTPSW\_PARAM\_MANIP\_POST\_TPSW2\_Mytest\_V /\* No pointer to copy: nothing to do \*/

RBTPSW\_DEF\_FUNC\_NORET\_1PARAM(MYTPSW2, 100, 0, RBTPSW\_NO\_TIMELIMIT, TPSW2\_Mytest\_V, uint8\_t, variable1)

#### LCF section configuration

The LCF section for both TPSWs looks like this:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* MYTPSW1\_LCF \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

RBTPSW\_SECTION\_START(MYTPSW1\_LCF)

/\* LCF entry for TPSW .data inside GRAM \*/

.MYTPSW1\_DATA align(4) :

{

RBTPSW\_MyTPSW1.o (.data)

RBTPSW\_MyTPSW1.o (.sdata)

} > .

/\* LCF entry for TPSW .bss \*/

.MYTPSW1\_BSS align(4) :

{

RBTPSW\_MyTPSW1.o (.bss)

RBTPSW\_MyTPSW1.o (.sbss)

} > .

RBTPSW\_SECTION\_END(MYTPSW1\_LCF)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* MYTPSW2\_LCF \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

RBTPSW\_SECTION\_START(MYTPSW2\_LCF)

/\* LCF entry for TPSW .data inside GRAM \*/

.MYTPSW2\_DATA align(4) :

{

RBTPSW\_MyTPSW2.o (.data)

RBTPSW\_MyTPSW2.o (.sdata)

} > .

/\* LCF entry for TPSW .bss \*/

.MYTPSW2\_BSS align(4) :

{

RBTPSW\_MyTPSW2.o (.bss)

RBTPSW\_MyTPSW2.o (.sbss)

} > .

RBTPSW\_SECTION\_END(MYTPSW2\_LCF)

The TPSW code itself is in the modules RBTPSW\_MyTPSW1 and RBTPSW\_MyTPSW2. The Flash should contain a section for initialized data:

.CROM.MYTPSW1\_DATA CROM(.MYTPSW1\_DATA) align(4) ABS : > .

A look to the map files allows a verification of the allocation (MYTPSW1).

.MYTPSW1\_LCF\_ALIGN\_START fee89590 00000000 0 0000000

.MYTPSW1\_DATA fee89590 0000002c 44 0011cd8

.MYTPSW1\_BSS fee895bc 00000363 867 0000000

.MYTPSW1\_LCF\_ALIGN\_END fee89920 00000000 0 0000000

fee89590+000000 \_\_\_**MYTPSW1\_LCF\_start**

.MYTPSW1\_DATA fee89594+000004 \_TPSW1\_ActiveWaiting

.MYTPSW1\_DATA fee89598+000004 \_TPSW1\_ActiveWaitingCore0Ack

.MYTPSW1\_DATA fee8959c+000004 \_TPSW1\_ActiveWaitingCore1Ack

.MYTPSW1\_DATA fee895a0+000004 \_TPSW1\_Global\_u32

.MYTPSW1\_DATA fee895a4+000004 \_TPSW1\_Zero\_f

.MYTPSW1\_DATA fee895a8+000004 \_TPSW1\_Result\_f

.MYTPSW1\_DATA fee895ac+000004 \_TPSW1\_Zero\_u32

.MYTPSW1\_DATA fee895b0+000004 \_TPSW1\_Result\_u32

.MYTPSW1\_DATA fee895b4+000004 \_TPSW1\_ForceTestId

.MYTPSW1\_DATA fee895b8+000004 \_TPSW1\_ErrorCounter\_UL

.MYTPSW1\_BSS fee895bc+000190 \_TPSW1\_Table

.MYTPSW1\_BSS fee8974c+000190 \_TPSW1\_DataTable

.MYTPSW1\_BSS fee898dc+000020 \_TPSW1\_Tpsw2DebugCore0

.MYTPSW1\_BSS fee898fc+000020 \_TPSW1\_Tpsw2DebugCore1

.MYTPSW1\_BSS fee8991c+000001 \_TPSW1\_Tpsw2StateByApp

.MYTPSW1\_BSS fee8991d+000001 \_TPSW1\_Tpsw2StateByFunc

.MYTPSW1\_BSS fee8991e+000001 \_TPSW1\_Global\_u8

fee89920+000000 \_\_\_**MYTPSW1\_LCF\_end**

For initialized RAM data you usually see an additional entry like

.CROM.MYTPSW1\_DATA 000124e8 0000000d 13 0011cd8

#### Call of a TPSW function (FUNC)

The call of a TPSW function (FUNC) can be done as follow (getting the application state is optional):

rbtpsw\_debuginfo\_t DebugInfoMYTPSW1, DebugInfoMYTPSW2;

rbtpsw\_appstatus\_t AppStatus;

…

AppStatus = RBTPSW\_START(TPSW1\_Prc);

if(AppStatus != ACTIVE) DebugInfoMYTPSW1 = TPSWGetDebugInfo(MYTPSW1);

…

ret = RBTPSW\_CALL(TPSW1\_Mytest\_UL(var1, &struct1));

if(TPSWGetAppStatus(MYTPSW1) != ACTIVE) DebugInfoMYTPSW1 = TPSWGetDebugInfo(MYTPSW1);

…

RBTPSW\_CALL(TPSW2\_Mytest\_V(5));

if(TPSWGetAppStatus(MYTPSW2) != ACTIVE) DebugInfoMYTPSW2 = TPSWGetDebugInfo(MYTPSW2);

…

RBTPSW\_CALL(TPSW2\_Prc());

if(TPSWGetAppStatus(MYTPSW2) != ACTIVE) DebugInfoMYTPSW2 = TPSWGetDebugInfo(MYTPSW2);

### Configuration switches

#### RBFS\_TPSWCapsule

Switch name: RBFS\_TPSWCapsule

This switch configures the availability of the TPSW framework, which provides the APIs and functionality to integrate TPSW with a corresponding MPU capsules.

|  |  |
| --- | --- |
| **Value** | **Description** |
| RBFS\_TPSWCapsule\_OFF | TPSW Capsule turned OFF |
| RBFS\_TPSWCapsule\_SC3 | TPSW Capsule turned ON, enabling memory protection |
| RBFS\_TPSWCapsule\_SC4 | TPSW Capsule turned ON, enabling memory protection and timing protection |

#### RBFS\_TPSWTestsuite

Switch name: RBFS\_TPSWTestsuite

This switch configures the availability of the TPSW testsuite, which is only used for development purpose.

Hint: Take care - setting must be \_OFF in case of customer project or of course series builds! Therefore the switch will be configured per default to RBFS\_TPSWTestsuite\_OFF.

|  |  |
| --- | --- |
| **Value** | **Description** |
| RBFS\_TPSWTestsuite\_OFF | TPSW test suite OFF. Test suite code will not be compiled. |
| RBFS\_TPSWTestsuite\_ON | TPSW test suite ON. Test suite code will be compiled and activated. See section *5 TEST SPECIFICATIONS*. |

#### RBFS\_TPSWSVPRestrictions

Switch name: RBFS\_TPSWSVPRestrictions

This switch configures the availability of additional memory access restrictions which affect CPU supervisor mode. Additional CPU supervisor mode restrictions are:

* MPM.SVP = 1 (=> MPU is active in supervisor mode)
* whole RAM is configured non-executable by MPU region 0.
* Only Flash memory regions are configured executable.

|  |  |
| --- | --- |
| **Value** | **Description** |
| RBFS\_TPSWSVPRestrictions\_OFF | TPSW Supervisor Mode Restrictions turned OFF. This is the default. |
| RBFS\_TPSWSVPRestrictions\_ON | TPSW Supervisor Mode Restrictions turned ON. |

# REQUIREMENTS

## Functional Requirements

### Requirements within this SW Group

|  |  |
| --- | --- |
| **Unique requirement ID** | **Description (What?) / Reason (Why?, Who?)** |
| **REQ\_HSW\_TPSWCapsule\_0001** | Provide mechanisms of memory encapsulation of a TPSW in the way that TPSW code cannot (accidentally) manipulate memory outside a predefined valid range of TPSW data. (Accidental) manipulations of this kind have to be detected and lead to a safe state of the system. |
| *REQ\_HSW\_TPSWCapsule\_0001a* | In case of a manipulation of the above kind, it has to be possible that the rest of the system is unaffected i.e. possible to continue operation without the TPSW in case this is useful/possible from a functional perspective |
| *REQ\_HSW\_TPSWCapsule\_0001b* | Also enforce private stack protection for TPSW to make the RAM encapsulation complete. |
|  |  |
| **REQ\_HSW\_TPSWCapsule\_0002** | Allow a TPSW consisting of several runnable entities (processes) running in different task sharing a single RAM area for encapsulation. |
| *REQ\_HSW\_TPSWCapsule\_0002a* | In case an RE fails with an abort error, also avoid execution of all other REs of that TPSW application. |
| **~~REQ\_HSW\_TPSWCapsule\_0003~~** | ~~Provide a framework to allow TPSW to call trusted functions/services during the TPSW execution.~~ |
| *~~REQ\_HSW\_TPSWCapsule\_0003a~~* | ~~Allow TPSW to invoke mutual exclusion between REs in different tasks.~~ |
|  |  |
|  |  |

## Non-Functional Requirements

## Constraints and Environmental Conditions

|  |  |
| --- | --- |
| **Unique Constraint ID** | **Description** |
| **CON\_HSW\_TPSWCapsule\_0001** | Encapsulation is only provided for the given cyclic System9 task scheme. It does not support ECC tasks (i.e. tasks blocking on events) |
| **CON\_HSW\_TPSWCapsule\_0002** | A trusted full privileged wrapper code from Bosch side exists in a TPSW- and project-specific way. This handles the interface of the TPSW and the interaction with the Bosch ESP code correctly. It uses the TPSWCapsule described here to configure and invoke the TPSWCapsule for a given TPSW correctly. It cares for completeness of data encapsulation i.e. does not allocate any critical data from the Bosch SW within the TPSW accessible memory area but uses proper data copies/buffers instead and is designed and reviewed carefully in this respect. It also handles the faults reported by the TPSW Capsule properly. |
| **~~CON\_HSW\_TPSWCapsule\_0003~~** | ~~Number of parameters passed to TPSW service call is limited by SWI/SVC µC and compiler concept~~ |
| **CON\_HSW\_TPSWCapsule\_0004** | Granularity of memory protection area size is limited to hardware MPU features. I.e. a potential of 2 aligned to a similar size potential of two start address. |
| **CON\_HSW\_TPSWCapsule\_0005** | TPSW not allowed in interrupts for safety reasons. Whether the capsule supports this technically has to be evaluated in case. |
|  |  |

# KNOWN OPEN ISSUES

*Before going to describe the SW Group design and description, a clear description must be said here about the ‘known aspects of the design’ Like Items for farther improvements and/or Items for further clarifications.*

***Rules:***

* *Description of all Open items must possess relevant identifiers.*

|  |  |
| --- | --- |
| **Unique Open Issue ID** | **Description** |
| **OPI\_HSW\_<SW\_group>\_<XXXX>\_{free\_text}** | Write here description of this open issue. |
| **OPI\_HSW\_<SW\_group>\_<XXXX>\_{free\_text}** | \* Add more rows if required |

# TEST SPECIFICATIONS

To test the TPSW Capsule, a special SW build with modified SW was implemented. There is a test suite that includes basically two TPSWs. The following processes are scheduled (Note that the processes defined in that wrapper have to be attached to the task scheme using mergeproc entry points.):

* process RBTPSW\_MyTPSWCapsule\_TestExecution\_Proc() is called in the x2 task on the core on which the 1ms task runs. This ensures a high preemption count during the execution of the test suite.
* process RBTPSW\_MyTPSWCapsule\_PreemptedWaitingAndReset\_Proc() is called in the x8 task
* process RBTPSW\_MyTPSWCapsule\_MultiCoreWaiting\_Proc() is called in the x2 task on the opposite core of the RBTPSW\_MyTPSWCapsule\_TestExecution\_Proc()

Start of the testing:

1. RBTPSW\_MyTPSWCapsule\_PreemptedWaitingAndReset\_Proc() starts and enters into an active waiting loop

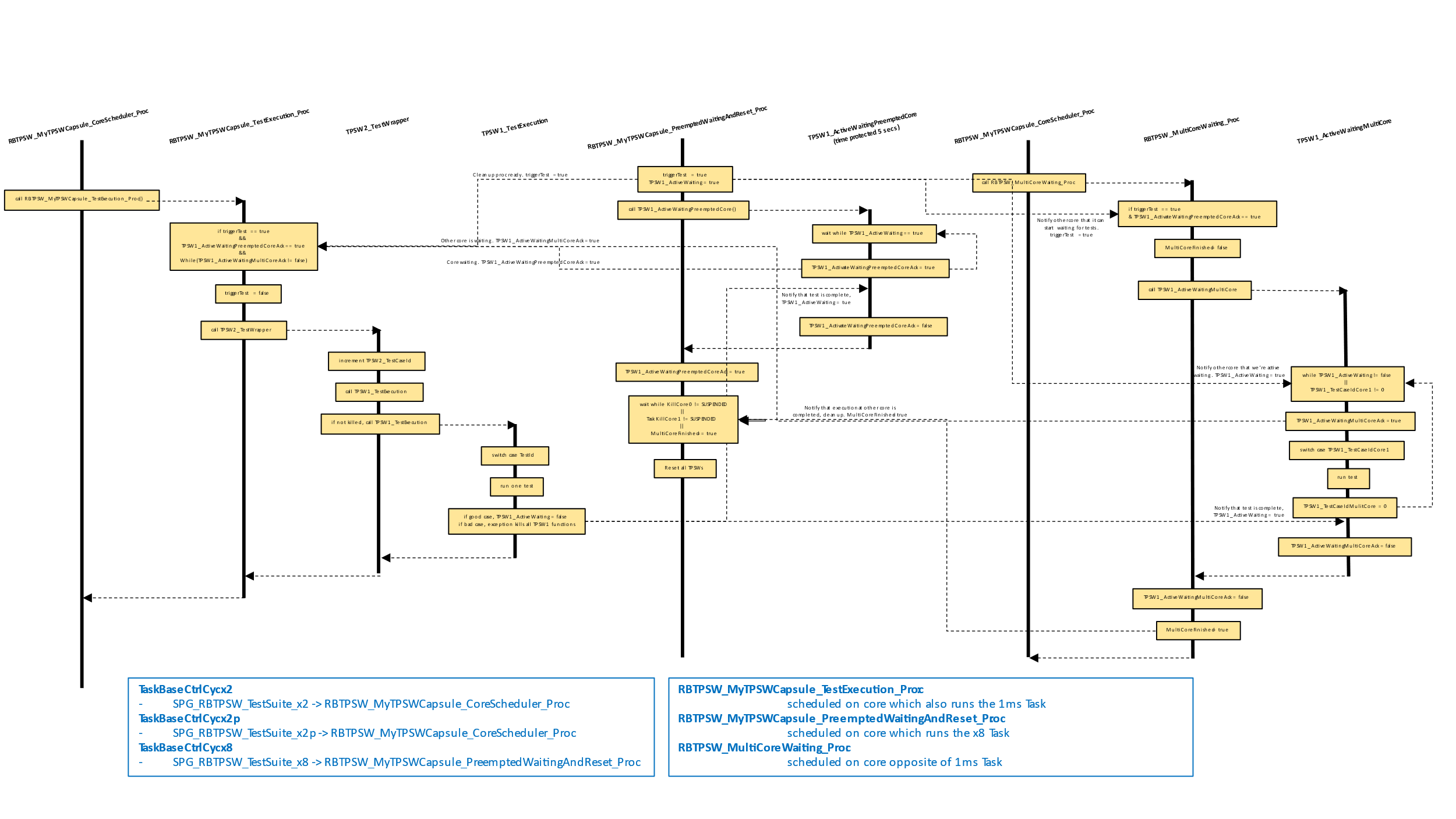
2. RBTPSW\_MyTPSWCapsule\_MultiCoreWaiting\_Proc() starts if running on a multi core system and only after RBTPSW\_MyTPSWCapsule\_PreemptedWaitingAndReset() has started. It also enters an active waiting loop.

3. RBTPSW\_MyTPSWCapsule\_TestExecution\_Proc() starts TPSW1 and the TPSW2\_TestWrapper if both RBTPSW\_MyTPSWCapsule\_PreemptedWaitingAndReset\_Proc() and RBTPSW\_MyTPSWCapsule\_MultiCoreWaiting\_Proc() have entered the active waiting loop. The actual test execution is performed in TPSW1\_TestExecution() which is called by the TPSW2\_TestWrapper().

End of testing in TPSW1:

1. RBTPSW\_MyTPSWCapsule\_PreemptedWaitingAndReset\_Proc() notifies RBTPSW\_MyTPSWCapsule\_MultiCoreWaiting\_Proc() and RBTPSW\_MyTPSWCapsule\_TestExecution\_Proc() that the active waiting loops can be stopped. See also *Abbildung 3: Good case test*
2. If TPSW1 was killed during the test (bad case), the active waiting loops will end when all cross core tasks are finished. See also *Abbildung 4: Bad case test*
3. If MYTPSW<x> was killed, it will be reactivated just before the new test cycle. Please note that this is done only for testing purpose. **It is forbidden to reactivate a killed TPSW in series software**

Sequence chart of TPSW test invocation: (also see Diagram\_TestFramework\_TPSW.pptx) 



Note:

* The overall test suite should not affect the rest of the SW although it continuously causes TPSW to be aborted or killed verifying that the TPSW’s misbehavior cannot affect the rest of the system. This can e.g. be seen on the task counters in the MT.
* The test suite’s results are made to be evaluated by MT.
* When measuring with MM6x, it is interesting to look at the following variables:
  + For applications:

RBTPSWAppCfg\_MYTPSW<N>.status

RBTPSWAppCfg\_MYTPSW<N>.PlausiCheckDone

RBTPSWAppCfg\_MYTPSW<N>.debuginfo[<core>].os\_status

RBTPSWAppCfg\_MYTPSW<N>.debuginfo[<core>].debug\_status

RBTPSWAppCfg\_MYTPSW<N>.debuginfo[<core>].userinfo

RBTPSWAppCfg\_MYTPSW<N>.debuginfo[<core>].FaultAdr

RBTPSWAppCfg\_MYTPSW<N>.debuginfo[<core>].CallCoreFnPtr

RBTPSWAppCfg\_MYTPSW<N>.debuginfo[<core>].rh850\_mei

RBTPSWAppCfg\_MYTPSW<N>.debuginfo[<core>].rh850\_mea

RBTPSWAppCfg\_MYTPSW<N>.debuginfo[<core>].rh850\_feic

RBTPSWAppCfg\_MYTPSW<N>.debuginfo[<core>].rh850\_fepsw

* + For good cases (both variable are equal to 0 if the test suite succeeded):

TPSW1\_ErrorCounter\_UL

TPSW2\_ErrorCounter\_UL

x2 task

TPSW1

x8 task

TPSW1

x2 task

TPSW1

TPSW2

**TPSW1**

**Test**

**execution**

Waiting until test is finished

Waiting until test is finished

Abbildung 3: Good case test

This strategy allows each test case to be executed in a cascading context (TPSW1 preempts TPSW1 with higher priority). Moreover the test execution happens during TPSW1 runs in parallel in another core. In this way following features can be tested when TPSW1 is killed:

* cascading abort
* cross core kill

x2 task

TPSW1

x8 task

TPSW1

x2 task

TPSW1

TPSW2

**TPSW1**

**Test**

**execution**

Waiting until test is finished

Waiting until test is finished

**Cascading abort**

**Cascading abort**

**Cross core kill**

Abbildung 4: Bad case test

Important test parameters:

* TPSW2\_TestId: current test case id to be performed. It is automatically incremented at each test period.
* TPSW1\_ForceTestId: overrides the TPSW2\_TestId and forces a test case to be executed.

The test case specification can be found under:

[\\bosch.com\dfsrb\DfsDE\DIV\CS\DE\_CS$\Prj\BaseDev\BSW\C\_Components\002\_OS\A\_AdvanceTopics\005\_CapsulationofBSW\CSMosar93\_MultiCore\doc\RBTPSW\_TestSuite.xlsx](file:///\\bosch.com\dfsrb\DfsDE\DIV\CS\DE_CS$\Prj\BaseDev\BSW\C_Components\002_OS\A_AdvanceTopics\005_CapsulationofBSW\CSMosar93_MultiCore\doc\RBTPSW_TestSuite.xlsx)

# DESIGN DECISIONS AND FACTORS

## TPSW capsule and OS integration

* No restart of an aborted TPSW allowed for safety reasons.
* An untrusted function cannot have any impact on the rest of the software (spatial and temporal). In other word, the TPSW framework must intercept any failure of an untrusted software through the Os abort / ProtectionLog mechanism.
* The whole framework is realized via “c” means, basically macros only. There was no need seen to involve code generators because all relevant use cases can be covered, tool qualification avoided and due to nature of encapsulating “c” code at any possible “c” call an adequate powerful low level approach was required.
* Use of RTA OS feature “call trusted function” in reverse way (extended feature from RTA OS) to call untrusted (TPSW) functions from a trusted (RB) task.
* There is only one Os untrusted application with one (un)trusted function per core configured in the OS. Handling of several TPSWs is managed via the framework only.
* One TPSW running in several tasks and cores via several FUNCs is allowed. ECC tasks (blocking tasks at events, which is in general not used at CC task scheme and much more complex) not supported.
* The usage of the framework in exception handlers and CAT 1 interrupts is not allowed (see details of implementation). This is because the TPSW framework relies on Os mechanism which can be only executed in Task or CAT2 ISR context.
* The interrupt lock and spin lock APIs for TPSW are realized basically as wrapper of existing locks with additional checks. Adding further locks with an additional (TPSW) lock counter would be too complex.
* Every TPSW failure must lead to a call of the Os ProtectionLog() function. That’s why all switch from user mode to supervisor mode explicitly (e.g. “trap 0”) or implicitly (e.g. via Os APIs) are preceded by a stack, relevant register and lock plausibility. This is done to ensure that no TPSW lead to an exception later in supervisor mode. (see also the “*Things to care about for a trusted to untrusted transition*” point in section 9.1.2)
* The TPSW relies on an OS feature called Enhanced Isolation. This feature introduces additional system checks when switching modes or when an interrupt is triggered – e.g. stack pointer check or global register reset. When a fault is detected by the OS in one of these checks, the Os\_CallTrustedFunction (internal API which protects and executes the untrusted function) is aborted and the TPSW is marked as faulty. Next the framework attempts to restart the FUNC, while this time configuring the MPU to make flash non-executable. This will cause an MPU failure and the rest of the abort will happen just as a normal exception, including any cascading and cross-core aborts.

The user callbacks required and called by Enhanced Isolation are implemented already in TPSW as:

* + FUNC(ProtectionReturnType, OS\_APPL\_CODE) **Os\_Cbk\_IsUntrustedCodeOK** (Os\_EIContextBuffType \* Os\_stack\_context, Os\_UntrustedContextRefType Os\_EIApplicationContext)
    - checks whether untrusted code has misbehaved or not
    - checks if r3 (sp) is okay
  + FUNC(boolean, OS\_APPL\_CODE) **Os\_Cbk\_IsSystemTrapAllowed** ( MemoryStartAddressType Caller)
    - allows or forbids trap 0
  + FUNC(void, OS\_CALLOUT\_CODE) **Os\_Cbk\_RestoreGlobalRegisters** (void)
    - called to restore r4 (gp) and r5 (tp)
  + FUNC(ProtectionReturnType, OS\_APPL\_CODE) **Os\_Cbk\_IsUntrustedTrapOK** (MemoryStartAddressType Os\_ret\_addr, uint32 Os\_CauseCode)
    - checks whether to allow or not a trap called from untrusted context to be executed

More information about this feature and a comprehensive description of the failure modes that it covers can be found on the OS documentation for Target Ports RH850/RH850x2.

## Memory protection

* Memory protection is applied for each untrusted function.
* RAM write is only avoided in new version of TPSW capsule. RAM read is allowed to allow simpler integration and no safety effect.
* Peripheral read and write is avoided.
* Calling a trusted FUNC or an untrusted FUNC is handled via the same API RBTPSW\_CALL/RBTPSW\_START in the framework.
  + In case a trusted function is to be called, the framework basically switches to trusted mode (and performs checks before and after). The CallTrustedFunction() call API of the OS is not used in that case (but would also work) for performance and simpler implementation.
  + In case a non-trusted function is called, a call to one OS CallTrustedFunction() is done but for the untrusted application. The correct TPSW is determined via parameter passing to Os\_Cbk\_SetMemoryAccess(). By this, several TPSWs are supported. A TPSW FUNC specific “core” function (generated via the configuration macros) is called then in untrusted mode. Parameter handling is done there and potential manipulators called. Then the real TPSW function is called. Afterwards, results and return values are copied back (and post manipulators called under trusted rights within the core function.
* RTA OS uses a single stack policy even for memory protected systems (SC3). So there is no separate stack used for TPSW but TPSW nests on the large system stack. Instead MPU channels are configured in a OS callout on the stack to open up a window on the stack from the current stack position to the required size when TPSW is called (see “MPU setup for stack” in the “detailed design” chapter).

## Timing protection

* Timing protection is optionally selectable for each untrusted function
* Autosar provides two different mechanism to reach the timing protection:
* *Passive timing protection*: the net runtime of a function is monitored at each preemption and at termination of the function. If the monitoring detects that the net runtime is bigger than the allowed budget, the Os reports a timing violation. The detection of the budget expiration **is not instantaneous**.
* *Active timing protection*: before the function start, a timer interrupt is programmed at the execution budget end. The timer deadline is shifted at each preemption to ensure the net execution protection. If the function budget’s expires, the timer interrupt will be fired immediately and the Os will report a timing violation. The detection of the budget expiration **is instantaneous**.

The TPSW framework is using the active timing protection mechanism as it is ensuring that an untrusted function will be killed as soon as its budget expires and will not influence other running applications. This ensures a clean temporal separation of the safety level.

* The interrupt lock time and spin lock time utilized by a timing protected function will not be monitored (neither actively nor passively). If the untrusted software utilizes more runtime than allowed, the RBSYS will react consequently.
* The timing protection interrupt is implemented as CAT 2 interrupt with interrupt priority level same as the lowest CAT 1 interrupt priority level which means it has the same operating priority as the OS code. As a consequence, it cannot interrupt OS code. This behavior is by intention:
  + If implemented as CAT 1 there were two serious issues:
    - Determinism: if the timing protection interrupt can interrupt the OS code at any stage, it is almost impossible to verify correctness of functionality by covering all cases by testing. For the sake of reducing the required test parameter space it is chosen to be safe not being able to interrupt OS code.
    - In case of nested timing protected calls there is an issue which detecting the timing abort of the outer TPSW FUNC. CallTrustedFunction calls two callbacks: one to start the timer and the second to set up the MPU environment. Only the latter passes information about the TPSW application. If the time budget is zero (e.g. after aborting an inner TPSW FUNC) and the timer interrupt is CAT1, it would call ProtectionHook without setting up the current TPSW application and lead to wrong cleanup behaviour
  + Implementing as CAT 2 has one issue:
    - If untrusted TPSW FUNC were allowed to block interrupts, the timing interrupt had no chance to interrupt an interrupt lock taken forever. It was chosen to overcome this by not allowing interrupt locks in timing protected functions.

# ALTERNATIVE SOLUTIONS

There would be another approach of using individual stacks for each TPSW as it was in the TPSW solution with RTA OSEK. This implies a lot of modifications and will not be compliant with further features used in RTA OS. So it was decided to follow the single stack approach of RTA OS even with TPSW.

Different TPSWs and FUNCs can also be handled by different Trusted Functions and different OS applications. But this would require a reconfig and regenerate of the OS. It was decided to avoid this solution for project handling reasons.

# DETAILED DESIGN DESCRIPTION

## Design Overview

See UML for structure details.

## SW group’s internal structure

The TPSW capsule is realized as a module with source and header file. The TPSW service is realized as an additional (optional) module. See UML for structure details.

## Internal Interfaces

See UML for structure details.

## Internal Data Structures

See UML for structure details.

## Internal Functions used

See UML for structure details.

# IMPLEMENTATION DETAILS

### MPU setup for Stack area

The MPU is used in terms of AUTOSAR i.e. applying it to complete tasks (and even ISRs), also allowing to separate OS applications in AUTOSAR terminology. The RTA OS uses a single stack area which is also to be applied for the TPSW framework.

The OS places a callout (Os\_Cbk\_SetMemoryAccess) to setup the MPU whenever a TPSW (task or function) is called. This callout is used to setup the MPU for normal .data .bss for TPSW and also for the stack area. So the callback return an offset to the OS to move the stack to the position the MPU can be set up when the TPSW is called.

This offset can be found in a very simple way, as the granularity of an MPU channel is 4 bytes alignment for start and end address. In this way it is possible to use a single MPU channel for the stack (a lot easier than the precedent generation with TMSx70).

Note: as the RH850 controller has a MPU channel granularity of 4 bytes, the stack repositioning could be avoided. However the TPSW framework shifts the stack of 8 bytes to force the repositioning feature to be used in the Os. This is done to avoid a known error in the “no stack repositioning” path of the Os. See also entry RB\_A218 of the Os tracking list (contact Os team @XC-ECO for more information).

The last address of the protected stack area is returned from the stack function and checked against the size of the full global stack in Os\_Cbk\_SetMemoryAccess(). The TPSW is aborted in case TPSW stack would exceed the global stack. This can happen if global stack size was not set correctly.

### Further details on implementation

These statements require code know how and deeper µC core HW understanding.

* “*Cascading abort*” mechanism
  + Realize cascading abort (abort of nested TPSW) via additional MPU channel that locks access to flash in case of an aborted TPSW.
  + Strategy: after killing a faulty TPSW, the SetMemoryAccess() callback will lock the flash for execution. When pre-empted FUNCs of the faulty TPSW are restored, they will be killed as soon as they try to execute something.
* “*Cross core kill*” mechanism
  + Strategy: when a running FUNC is killed, we have a core local way to kill all other FUNC from the current TPSW called “cascading abort”. We also need a mechanism which kills all other FUNC of this TPSW running on other cores. This is called “cross core kill” mechanism. Just after the faulty FUNC was killed and that the restoration happened, the TPSW framework will activate a high priority “*cross core kill task*” on all other cores which will pre-empt all running FUNC. This task force a CallTrustedFunction() call to ensure a reprogramming of the MPU channels. When this call ends, the SetMemoryAccess() callback will use the “cascading abort” mechanism to kill all FUNC of each core local cores. See also *Abbildung 5: Cross core mechanism.*

Note: the “*cross core kill task*” multiple activations, as several abort can occur in a very small amount of time. This ensures that all cross core kill requests are taken in account. An alternative to the cross core task would be a cross core interrupt, but it is not possible to generate multiple interrupt requests.

* + Limitation: **the cross core kill mechanism will only kill TPSW running on a task and not in CAT2 ISR.**

Core 0

Core 1

Task B

TPSW1, Function B

Task A

TPSW1, Function A

Cross core 1

kill task

Cross core kill

UT application

1. The function A of TPSW1 is faulty. An exception is generated: it kills TPSW1 and aborts the function.

2. The abort handler activates a highest priority task on core 1.

3. A dummy untrusted application is started to force MPU reprogrammimg

4. Cascading abort of Function B

Cross core 0

kill task

Cross core kill

UT application

5. The abort handler activates a highest priority task on core 0. This is a side effect of the cross kill mechanism and can be ignored.

Abbildung 5: Cross core mechanism

* Data collection during execution:
  + We hold a static variable “current\_tpsw\_app\_cfg[]” that contains the last application for which Os\_Cbk\_SetMemoryAccess has been called per core. This is needed because GetApplicationID() returns ESPapp always when using fcn-level protection
  + Furthermore the last core function is set in a global variable “current\_tpsw\_fncall\_core[]” per core to have it available as debug value in case TPSW aborts (to help to find the causing FUNC).
* The RH850 devices have 16 MPU channels. Currently the P1x device uses 11 of them and the U2A uses 16 of them:  
  P1x:
  + Region 0: 4 GB region: SVP\* deactivated (MPM.SVP==0) => not needed.   
    SVP activated (MPM.SVP==1) => supervisor rw
  + Region 1: flash bank A\*\*: (user rx, SVP: rx) for TPSW kill and const read
  + Region 2: flash bank B: (user rx, SVP: rx) for TPSW kill and const read

*(there is an address gap between flash bank A and B, therefore 2 regions must be used)*

* + Region 3: whole global ram (user r-only, SVP: r-only)
  + Region 4: whole local ram0 (user r-only, SVP: r-only)
  + Region 5: whole local ram1 (user r-only, SVP: r-only)
  + Region 6: stack (user rw, SVP: rw)
  + Region 7: mpu table entry 0 (used if configured) (user rwx, SVP: rw)
  + Region 8: mpu table entry 1 (used if configured) (user rwx, SVP: rw)
  + Region 9: mpu table entry 2 (used if configured) (user rwx, SVP: rw)
  + Region 10: mpu table entry 3 (used if configured) (user rwx, SVP: rw)

U2A:

* + Region 0: 4 GB region: SVP deactivated (MPM.SVP==0) => not needed.   
    SVP activated (MPM.SVP==1) => supervisor rw
  + Region 1: flash bank A: (user rx, SVP: rx) for TPSW kill and const read
  + Region 2: flash bank B: (user rx, SVP: rx) for TPSW kill and const read

*(flash banks A to D are consecutive in the address map of U2A)*

* + Region 3: whole global ram (user r-only, SVP: r-only) – CRAM0
  + Region 4: whole local ram0 (user r-only, SVP: r-only)
  + Region 5: whole local ram1 (user r-only, SVP: r-only)
  + Region 6: stack (user rw, SVP: rw)
  + Region 7: Cluster RAM 1 (user r-only, SVP: r-only) - CRAM1
  + Region 8: Cluster RAM 2 (user r-only, SVP: r-only) - CRAM2
  + Region 9: Cluster RAM 3 (user r-only, SVP: r-only) - CRAM3
  + Region 10: flash bank C (user rx, SVP: rx) for TPSW kill and const read
  + Region 11: flash bank D (user rx, SVP: rx) for TPSW kill and const read
  + Region 12: mpu table entry 0 (used if configured) (user rwx, SVP: rw)
  + Region 13: mpu table entry 1 (used if configured) (user rwx, SVP: rw)
  + Region 14: mpu table entry 2 (used if configured) (user rwx, SVP: rw)
  + Region 15: mpu table entry 3 (used if configured) (user rwx, SVP: rw)

\*Note 1: ‘SVP’ means that RBFS\_TPSWSVPRestrictions == RBFS\_TPSWSVPRestrictions\_ON or supervisor mode restrictions turned on – RAM not executable.

\*\*Note 2: In case FOTA (firmware-over-the-air) is used, some additional restrictions are applied in the MPU configuration at initialization and during runtime:

* If Enhanced FOTA is enabled, Flash Bank A may be erased and reprogrammed during run time and therefore no read/write/execute access is allowed for TPSW.
* Peripheral address range is checked internally. If a configured area start address is larger than 0xFF000000, it is assumed to be a peripheral and “device settings” are applied for access. Addresses lower than 0xFF000000 are considered to be accesses with “normal” memory settings.
* The following exceptions are mapped to Os\_abort():
  + MIP/MDP: MPU exception
  + PIE: Access to supervisor registers

The following exceptions will call RBSYS if they come from supervisor mode, else they will call Os\_abort():

* + RIE: reserved instruction exception
  + MAE: alignment error
  + FPP: floating point exception. This is a EI level interrupt. If it was generated from user mode, it will program the flash mpu and the used RAM channels for read only (no execution) and returns. As soon as it has return, the program execution will generate a MPU error.

The Os\_abort() handler will get a safe stack for the case the stack pointer was broken by the faulty TPSW (“Get abort stack mechanism” from Os). It will then call the ProtectionLog() and ProtectionHook() were:

* + the faulty TPSW will be killed
  + all locks will be reset
  + the debug structure will be filled

After this the ProtectionLog() performs a LongJmp() to restore the CPU to its original state. The PC is back to the CallTrustedFunction() API. It will care about restoring the interrupt level.

* The Autosar standard forbids calling any Os API under interrupt lock. As the TPSW framework relies on Os mechanism, it is also forbidden to call any TPSW API under interrupt lock. Only exception: “exit lock” APIs. The TPSW framework checks all interrupt counters before all TPSW API and kills a TPSW if it uses under interrupt lock.
* The Intlock during the Inittask was removed for the same reason. Usage of TPSW framework in inittask would release intlock unintentionally. We prefer to have TPSW framework available in inittask.
* Strategy for checking if locks are active (see also RBTPSW\_AreCoreLockActive()):
  + Single core: check the core local interrupt lock counter
  + Multi core: Only need to check if the core local interrupts are locked. This will also covers the common lock/nonest locks which are always under core local int lock.
* TPSW framework cannot be used in a CAT1 ISR because TPSW calls CallTrustedFunction that sets IntlockMask to CAT2 level (is lower than CAT1 => reactivation). Therefore all ISRs were converted to CAT2 interrupts.
* We checked the stackoverhead of the framework for small functions and set it to minimum 64 bytes in the framework internally.
* A structure with **pointers** to all params is passed from the macro generated Call frame to the untrusted function through the OS \*Param structure parameter of a CallTrustedFunction. A pointer is enough here. In the Core function, which is executed with the rights of the callee, a copy of all parameters is done (read from stack of caller is allowed).
* Pointer checks for validity of passed pointers from TPSW => Trusted or TPSW(UT) => TPSW (UT) have to be done in the individual TPSW specific wrapper by the integrator by e.g. forced access on caller side. It cannot be checked by the framework because signature, addresses and ranges are not available to the framework. Maybe a further redirection is needed.
* If a trusted function causes interrupt (asymmetric locks) or stack problems, this is not explicitly checked by the TPSW framework (runtime issue) but have to rely on the general platform monitoring for this.
* Things to care about for an untrusted-to-trusted transition:

Before each untrusted to trusted transition, all relevant variables and register which can influence the trusted software must be checked. These are the relevant transitions:

* + API:
    - Leaving a TPSW function (back to the RBTPSW\_FnCall\_Core\_##fnname() function)
    - Entering RBTPSW\_CALL()
    - RBTPSW\_Switch2PrivMode()
    - Enter/Exit spinlock and interrupt lock
  + Interrupt pre-emption of an untrusted software
  + Os\_Abort() caused by an untrusted software

These are the relevant registers/ variables to check:

* + Stack pointer: following mechanism are used:
    - Force stack and stack plausibility (used in APIs)
    - Untrusted stack check (Interrupt pre-emption)
    - Get abort stack (used in abort handler)
  + Text pointer, Global pointer: are reset to the value start-up value defined by RBSYS before each switch to trust. Exception: switch to Exceptions/ISRs/Os APIs: will be handled later by the Os.
  + Working registers: a broken stack can lead to the usage of broken working register in trusted code. If these registers are used as local variable values (e.g. pointer, ... ), this can have a safety impact. The relevant local variables are checked for consistency by storing their inverted values.
  + Interrupt lock and spin lock counter
* Suppression flag mechanism

# ANNEXURE

## Annexure A

## Annexure B

# FURTHER DEVELOPMENT

Pot. Timing protection in the sense of killing an application when a budget is over which has to be developed carefully considering available timer and compliance with safety concept implementation details (e.g. FIQs are not allowed for this feature to use because exclusively reserved for µC safety,…).

## Items Planned

|  |  |
| --- | --- |
| **Item** | **Description** |
| **Simple Name** | Write here description of this item. |
|  | \* Add more rows if required |

## Items OnHold

|  |  |
| --- | --- |
| **Item** | **Description** |
| **Simple Name** | Write here description of this item. |
|  | \* Add more rows if required |

# FEATURE AND RESSOURCE MANAGEMENT

*This chapter gives an estimation of resource consumption of the features implemented by the SW group and measures to reduce it.*

***Rules****:*

* *Write down here a rough estimation of how much ROM, RAM and runtime your SW group will consume, when all features are implemented. Fill in the list below.*
* *Give some suggestions of how resources could be reduced* 
  + *Example: Some features not really needed for low end ABS could be switched off. Is there a splitting of functionality by function switch possible? (Example switching off StdSPI for ABS)*
  + *Are there some features (for example features related to diagnosability like fault type tests, debug data) that could be deactivated for a low-end configuration?*
* *Give a hint to what extent your code is resource optimized up to now. (Rate this with an (--) to (++) for your main features.*

## Resource consumption of missing features

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Item** | **Feature** | **ROM (K)** | **RAM (K)** | **Runtime (%)** |
| **Filename** | Write down here names of features/requirements which are still missing in this module | Write down here rough estimation of ROM consumption | Write down here rough estimation of RAM consumption | Give a hint if you expect a significant runtime adding effect |
| \* Add more rows if required |  |  |  |  |

## Possibilities to reduce resources