TrafficLight: Arm FuSa RTS process isolation example

FuSa RTS Tutorial

AN336, Summer 2021, v1.1

keil-feedback@arm.com

Abstract

This application note demonstrates how to use the process isolation capabilities present in Arm's **Functional Safety Run-Time System** (FuSa RTS) [1].

The **TrafficLight** example described in the application note supports these boards out of the box: Keil MCBSTM32F400, STMicroelectronics NUCLEO-F429ZI and NUCLEO-F746ZG. However, the analysis and principles are mostly universal and can be similarly applied to other Cortex-M devices with a Memory Protection Unit (MPU) present.

The example project is provided as part of commercial FuSa RTS deliverables. Contact Arm Sales or local Arm distributor to receive it for evaluation purposes.

Prerequisites

To reproduce the example described in this application note, the following components are required:

Components from Arm:

- **Arm FuSa RTS** for Cortex-M4 or M7: run-time system for functional safety applications. Verified with FuSa RTS v1.1.0.
- **Arm Keil MDK**: IDE and debugger used for project development and debug. MDK v5.35 was used.
- **Arm Compiler for functional safety**: safety-qualified C/C++ compiler for Arm devices. Required by Arm FuSa RTS and available with the **MDK-Professional** edition. Version 6.6.4 is used.
- **Keil MDK-Middleware**: included in Keil MDK. Middleware v7.13.0 was used to verify the project.
- **Arm CMSIS-Zone utility** (optional): a graphical tool that allows to define partitions (zones) for a target embedded system. CMSIS-Pack Eclipse Plugin v2.6.0 is used in the project.
- **Arm CMSIS**: standardized software interfaces for Cortex-M. The referenced example uses CMSIS-Driver API components. The application is verified with CMSIS v.5.8.0.
- **Keil::STM32F4xx_DFP** or **Keil::STM32F7xx_DFP**: Device Family Pack (DFP) for target STM32F4 or STM32F7 devices respectively. Among other items, it contains the device header files and HAL used by the application. Verified with Keil::STM32F4xx_DFP v2.15.0 and Keil::STM32F7xx_DFP v2.14.0.
- **Keil::STM32NUCLEO_BSP**: Board Support Pack (BSP) for STM32 NUCLEO boards. Verified with version 1.8.0.
- **Keil MCBSTM32F400** development board with STM32F407IG Cortex-M4 based microcontroller. Version 1.2 is used by default, and v1.1 can be used with simple modifications in the project.

Components from ST:

- **X-CUBE-STL** (optional): software test library for target STM32F4 or STM32F7 devices. V1.0.0 is used.
- **STM32CubeProgrammer** (optional): programming utility for STM32 devices. Required when X-CUBE-STL is enabled in the project. Version 2.6.0 is used.

  See **AppNote 326: Using X-CUBE-STL with Arm FuSa RTS [5]** for further details.
- **NUCLEO-F429ZI** or **NUCLEO-F746ZG** development board with STM32 Cortex-M4 or Cortex-M7-based microcontroller. Can be used as one of the hardware targets for the example application.

Contents

Introduction

Some functional safety standards require a thorough analysis of the software architecture on different levels. This presents developers with great challenges. On the one hand, it is required to prove that all required software elements are available, effective, and developed according to the correct ASIL (Automotive Safety Integrity Level). On the other hand, a second required safety analysis needs to consider the dependent errors. It serves to prove that all components are independent of one another without influencing each other.

To make these analyses easier, especially in a single-core embedded system (executing functionalities of different safety integrity levels with different safety requirements in a single core), process isolation can be sued to ensure that the non-safety part (or the part with lower integrity level) does not impact the operation of the safety critical part (or the part with higher integrity level) of an application.

This application note introduces the process isolation capabilities of Arm FuSa RTS that enable such a protection from interferences. Implementation of process isolation is then explained using an exemplary traffic light application.

The application note is structured as follows:

- **Introduction** outlines the structure and scope of the application note.
- **Arm FuSa RTS overview** introduces Arm FuSa RTS and its process isolation capabilities.
- **TrafficLight: A process isolation example** explains the implementation of the TrafficLight example.
- **References and useful links** are provided related to the topic of this application note.

Scope of the application note

The TrafficLight example demonstrates how process isolation can be used in an Arm FuSa RTS based application.

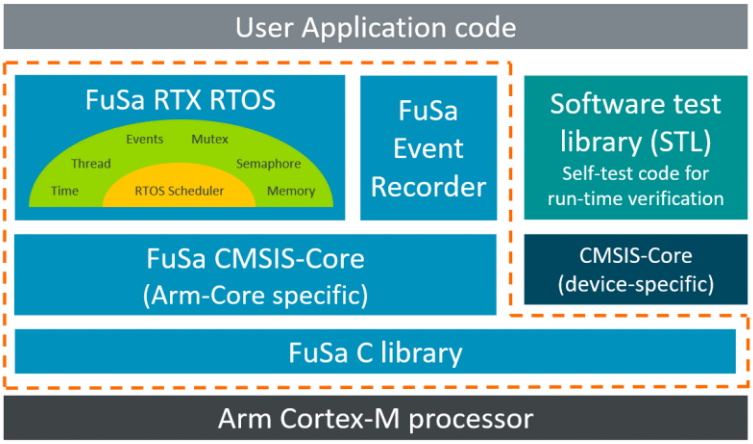
Example is provided for specific target devices. However, the concepts are universal and can be similarly applied to other Cortex-M devices with a Memory Protection Unit (MPU) present.

This application note provides an application example for explanatory purposes to show use of process isolation concepts and key APIs. To keep the example simple, many FuSa RTS user safety requirements are explicitly omitted as mentioned in **Safety requirements**. This application note is not part of the FuSa RTS Safety Package.

The FuSa RTS Safety Package is the main reference for the user implementing safety-related systems with process isolation on FuSa RTS.

Arm FuSa RTS overview

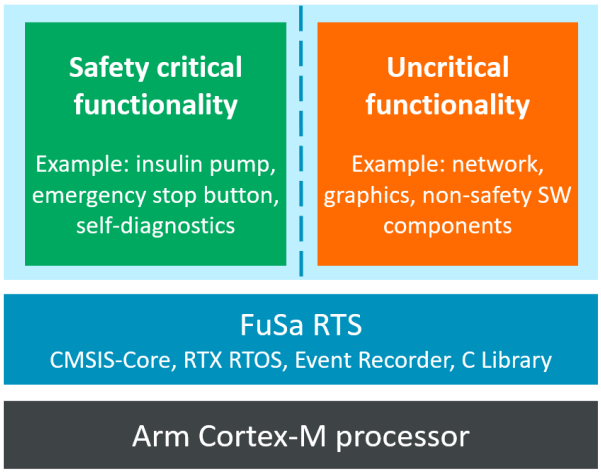
Arm's Run-Time System for Functional Safety (FuSa RTS) is a set of safety-certified software components for Cortex-M based devices. It contains a real-time operation system (FuSa RTX RTOS), the processor abstraction layer (FuSa CMSIS-Core), the FuSa Event Recorder and the FuSa C library. The system block diagram below shows all FuSa RTS components (blue blocks within the red dotted line):



Arm FuSa RTS in an embedded application

Process isolation in FuSa RTS

Process isolation functionality is introduced in FuSa RTS 1.1.0 and allows to achieve freedom from interference required for execution of functionalities with different integrity levels:



FuSa RTS for software components with different safety levels

The access to system resources (such as memory, peripherals, processor execution time) should be controlled to avoid undesired mutual interferences between software elements of different integrity levels (with different safety requirements). This is achieved in FuSa RTS with a following functionalities:

- **Spatial isolation** shields access to microcontroller memory or peripherals. In FuSa RTS this is enforced by MPU Protected Zones and Safety Classes.
- **Temporal isolation** is the ability of a system to respect its own timing constraints. This is supported with thread watchdogs that get triggered if timing requirements are violated.
- **Controlled system recovery** allows to control system operation in case of failures and so block execution of non-safety critical parts or proceed to a safety state.

Detailed description of process isolation in FuSa RTS is provided in FuSa RTS Safety Manual.

TrafficLight: A process isolation example

The example project implements a simple traffic light controller with standard traffic light phases (red, red/yellow, green, yellow, red). The time interval for green phase depends on the emulated traffic value.

The process isolation capabilities of FuSa RTS are fully utilized in this example to demonstrate how to protect components with different safety levels from undesired mutual interference.

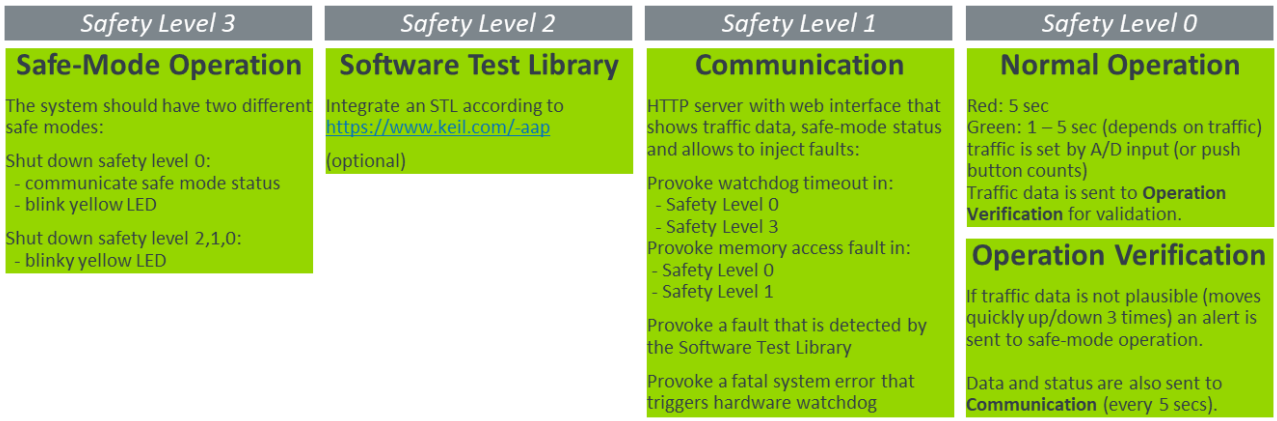
The control component **Normal Operation** is executed by default and has low safety integrity level. The operation of the control application is monitored (**Operation Verification**) for plausible operation; and in case of incorrect operation this is reported to "**Safe-Mode Operation**" with high safety integrity which then brings the system into a safe state. In the Safe-Mode Operation only the yellow light blinks.

The **Communication** interface (in this example TCP/IP stack over Ethernet) allows to receive user input and provides status information via hosted webserver.

In case of MPU faults or thread watchdog alerts caused by Safety Level 0, only the threads of the same safety level are suspended. This still allows to maintain network communication that has Safety Level 1.

When a fault is caused by higher safety levels, then all functionality with lower safety levels is suspended and only the safe state operation remains.

Figure below briefly describes operations at different safety levels implemented in the TrafficLight example.



Application overview

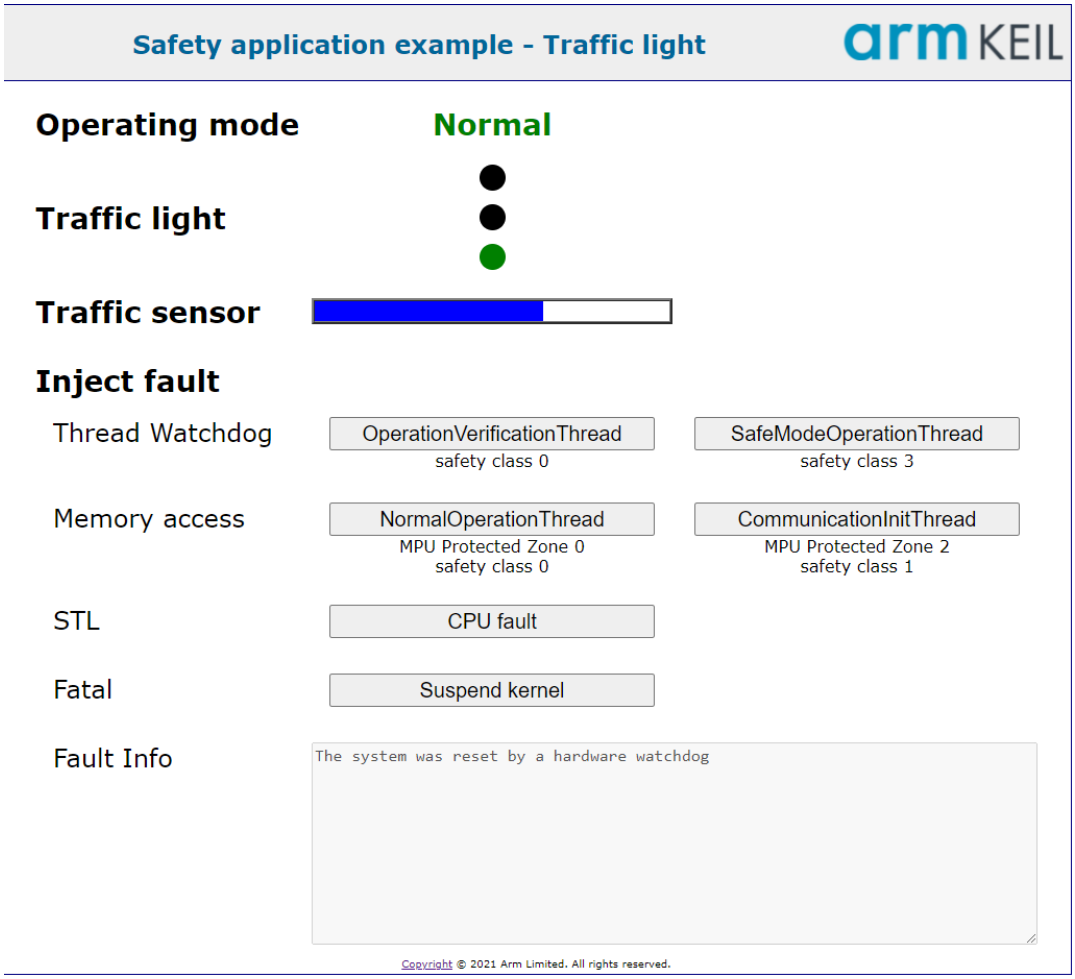
The table below explains the application functionalities with the corresponding threads that implement them, as well as assigned Safety Class and MPU Protected Zone (including memory regions accessible from it).

Functionality	Safety Class	MPU Protected Zone		
		Flash regions	RAM regions	I/O regions
Normal operation *SensorThread* reads ADC value from potentiometer emulating traffic data. *NormalOperationThread* switches between red and green lights (via yellow) with intervals depending on the measured traffic. A control loop in *OperationVerificationThread* verifies the traffic data on plausibility. The data and status are periodically sent to an HTTP server.	0	**Zone 0 (ZONE_NORMAL_OP):** ***NormalOperationThread, SensorThread***		
		CODE (Flash): p/u: RO, X	RAM_NORMAL_OP RAM_SHARED RAM_EVR p/u: RW, XN	ADC GPIO p/u: RW, XN
		Zone 1 (ZONE_VERIFY_OP): ***OperationVerificationThread***		
		CODE (Flash): p/u: RO, X	RAM_VERIFY_OP RAM_SHARED RAM_EVR p/u: RW, XN	None

Functionality		Safety Class	MPU Protected Zone		
			Flash regions	RAM regions	I/O regions
Communication HTTP web-server that gets accessed remotely over TCP/IP connection with Ethernet interface. Displays traffic data, status. Can be used to inject various faults to the system. Uses MDK-Middleware Network stack. Also see **Usage of the MDK-Middleware network** component		1	**Zone 2 (ZONE_COM):** ***CommunicationInitThread, netCore_Thread, netETH0_Thread***		
			CODE (Flash): p/u: RO, X	RAM_COM RAM_SHARED RAM_EVR p/u: RW, XN	SYSCFG, EXTI GPIO, RCC, Ethernet p/u: RW, XN
Software Test Library (optional) Integrates Software Test Library X-CUBE-STL for diagnostic tests. *StlThread* is executed in privileged mode. Also see **[5]**.		2	**Zone 3 (ZONE_STL)** ***StlThread*** *(privileged)*		
			CODE (Flash): p/u: RO, X	RAM_STL RAM_SHARED RAM_EVR p/u: RW, XN	None
Safe-Mode operation This is a safe state of the application. Yellow light is blinking indicating that normal operation of the traffic light is not possible.		3	**Zone 4 (ZONE_SAFE_OP):** ***SafeModeOperationThread***		
			CODE (Flash): p/u: RO, X	RAM_SAFE_OP RAM_SHARED RAM_EVR p/u: RW, XN	GPIO, HW WDOG p/u: RW, XN
RTOS operation	**Kernel** Also see **RTX kernel operation**	N/A	Not applicable as RTX Kernel runs in privileged mode and has no restrictions on memory access.		
	Timer Thread *osRtxTimerThread* is part of RTOS and shall not be modified. Also see **Timer configuration**	1	**Zone 5 (ZONE_TIMER):** ***osRtxTimerThread***		
			CODE (Flash): p/u: RO, X	RAM_TIMER RAM_COM RAM_SHARED RAM_EVR p/u: RW, XN	None
	Idle Thread *osRtxIdleThread* can be modified by user. Also see **Thread configuration**	4	**Zone 6 (ZONE_IDLE):** ***osRtxIdleThread***		
			CODE (Flash): p/u: RO, X	RAM_IDLE RAM_SHARED RAM_EVR p/u: RW, XN	None

Web interface

When the target board is available in the same LAN as the PC, its web page can be accessed from a browser using the board address (for example *https://mcbstm32f400*, *https://nucleo-f746zg* or *https://nucleo-f429zi*). Then following page showing status information and controls is displayed:



Operating mode shows the current mode of the application.

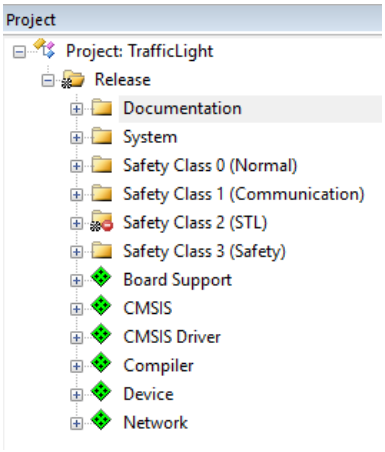
- **Normal**: switching between green and red lights (via yellow) with intervals depending on measured traffic. This is a default mode at system start.
- **Safe:** blinking with yellow light only. System is switched to safe mode when certain errors occur in normal operation mode. See **Fault handling** for additional information.
- **Traffic light** shows current light status green, yellow or red.
- **Traffic sensor** shows current emulated traffic density.
    - On the MCB board, the traffic is emulated based on ADC measurement from the on-board potentiometer. The data is considered implausible for high values. This can be triggered by turning the potentiometer clockwise to the limit.
    - On a NUCLEO board, traffic data changes when the USER button is pressed. A double-click is used to force an implausible value.
- **Inject fault** area contains multiple buttons that can be used to inject different faults to the system. Hovering over the button with a cursor will show additional information. Refer to section **Injecting** faults for further details.
- **Fault info:** displays the fault information reported by the device.

LED indications

On-board LEDs are used for emulating the traffic light behaviour with the following mapping:

On an MCB board		On a Nucleo board	
LED0	Red	LED1 (green)	Green
LED1	Yellow	LED2 (blue)	Yellow
LED2	Green	LED3 (red)	Red

Project organization in µVision



The application project in µVision is organized based on the functionalities.

The *System* group contains the main file, zone partitioning, fault handling, recovery, and other system-level items that are not part of any specific thread.

Other groups are made up of application files that implement specific functionality and are named based on the safety class assigned to them (for example *Safety Class 0 (Normal)*)

Software components used from the packs are displayed separately, even if they implement functionality for a specific safety class (for example *Network*).

Safety requirements

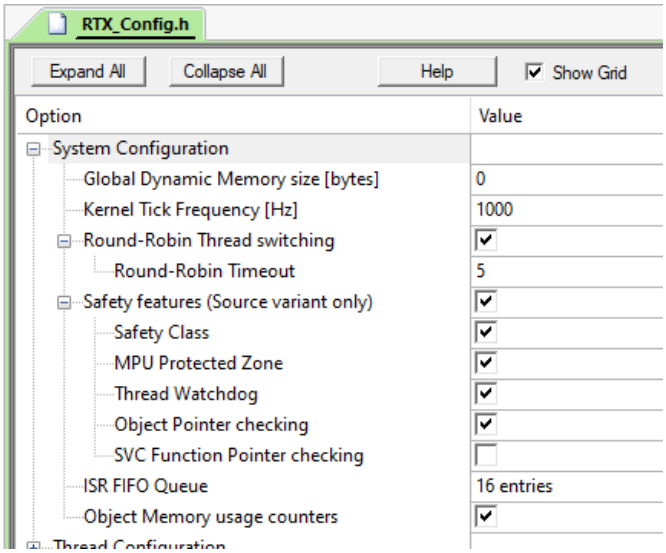
The TrafficLight example is intended to demonstrate the usage of process isolation capabilities in a complex application. To simplify the code and provide better readability, many of the FuSa RTS user safety requirements were explicitly omitted (for example verification of return codes), while others are provided only as template implementations (for example error handling routines). The FuSa RTS safety package is the main reference for users implementing applications based on FuSa RTS.

FuSa RTX configuration

FuSa RTX configuration parameters are defined in the *RTX_Config.h* file. It is recommended to use the Configuration Wizard view that provides a GUI-style approach for specifying the parameters.

System configuration

The figure below shows settings in the **System Configuration** group used in the TrafficLight example:



Global Dynamic Memory size is set to *0* as dynamic memory allocation is generally not recommended for safety reasons. Application uses object-specific memory allocation as well as static memory allocations.

Object Memory usage counters are enabled and can be used to verify the actual number of objects in use.

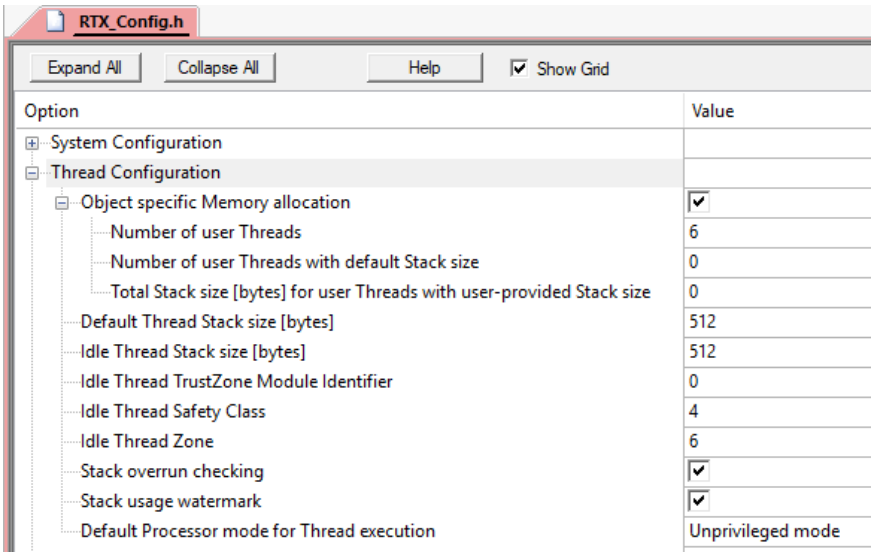
Safety features and all its sub-items except **SVC Function Pointer checking** are enabled to ensure that process isolation features are available for use and operational.

SVC function pointer checking can be useful in safety applications. In the TrafficLight example, it is disabled to avoid additional complexity in the zone configuration.

Additionally, safety class and MPU Zone numbers are provided for the Idle thread (in the **Thread Configuration** group) and for the Timer thread (in the **Timer Configuration** group). The assigned values correspond to those listed in in **Application overview**.

Thread configuration

Figure below shows **Thread Configuration** group in *RTX_Config.h* file for the TrafficLight example.



Object specific Memory allocation is enabled for user threads. The **Number of user Threads** that can be simultaneously allocated in such a way is set to 6. The threads *netCore_Thread* and *netETH0_Thread* are statically allocated in the MDK-Middleware Network stack and do not need to be considered here. Idle and Timer threads are system threads and are also excluded from this number.

For user threads that are created with control block pointer set to NULL (*cb_mem* field in the *attr_bits*), the control blocks are automatically placed into the object-specific memory array allocated by the RTOS kernel (part of named section *.data.os.*). The *scatter.sct* file then maps this memory to the RAM_PRIVILEGED region that is not directly accessible from threads. Section **RTX kernel operation** gives additional details.

The *Number of user Threads with default Stack size* as well as *Total stack size [bytes] for user Threads with user-provided Stack size* are both set to 0 to disable thread stack allocation within the RTOS. Hence for each user thread a stack needs to be defined in the user code and placed into the memory region enabled for access from the MPU Protected Zone assigned to the thread. In our example all thread stacks are kept separately in the corresponding memory areas. Since RTOS does not need to allocate stacks for the user threads in our example, this configuration option is set to 0.

Idle thread

The Idle thread is created by the kernel according to the settings provided in the **Thread Configuration** group as shown in the figure above.

Idle Thread Safety Class is **4** – the highest among the threads in the TrafficLight example. This ensures that the thread cannot be suspended, as otherwise scheduling will not work correctly.

Idle Thread Zone is **6** (*ZONE_IDLE*) that includes following RAM regions:

- RAM_IDLE region for the thread stack and local variables as defined in the *scatter.sct* file:

```
RAM_IDLE REGION_RAM_IDLE_START REGION_RAM_IDLE_SIZE {
  * (ram_idle)
  rtx_lib.o (.bss.os.thread.idle.stack)
}
```

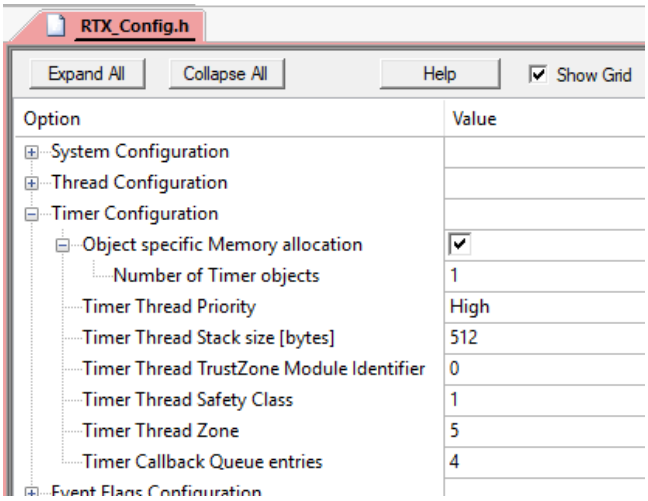
  ram_idle is a named memory section used in *RTX_Config.c* for mapping a local variable *wait_counter.* It is present just as an example to show how user variables can be mapped into thread memory.

- RAM_SHARED is configured as available for all threads. Not used by the Idle thread.

- RAM_EVR enabled for all threads to access Event Recorder. Not used by the Idle thread. See **Usage of FuSa Event Recorder**.
- RAM_PRIVILEGED and ARM_LIB_STACK regions are present as a work-around. If a region is not assigned to a zone, then it is not being put into the scatter file template by CMSIS-Zone. To avoid that, we enable RAM_PRIVILEGED and ARM_LIB_STACK for the Idle Thread zone but with privileged access only, thus they are not available to be accessed from the Idle Thread because it runs in an unprivileged mode.

Timer configuration

The figure below shows **Timer Configuration** group in *RTX_Config.h* file for the TrafficLight example:



The TrafficLight example uses 1 timer that is created in the MDK-Middleware Network component and relies on the object-specific memory allocation for it.

Timer Thread Safety Class is **1** – it is the same as for threads in the Communication zone where RTX Timer objects are used.

Timer Thread Zone is **5** (*ZONE_TIMER*) including the following RAM regions:

- RAM_TIMER for the thread stack, as defined in the *scatter.sct* file:

```
RAM_TIMER REGION_RAM_TIMER_START REGION_RAM_TIMER_SIZE {
 rtx_lib.o (.bss.os.thread.timer.stack)
}
```

  Section **Techniques for memory mapping** explains in an example how the timer thread stack is allocated.

- RAM_SHARED configured as available for all threads. Not used by the timer thread.
- RAM_EVR to access Event Recorder. See **Usage of FuSa Event Recorder**.
- RAM_COM required to enable Communication timer callback to access Communication variables.

Timer Callback Queue entries is kept at default value 4.

Event flags configuration

TrafficLight example uses 2 event flags. They are allocated via object-specific memory in RTX.

See section **Communication across safety classes** for more details on how the event flags are used.

Message queue configuration

One message queue object is used in the TrafficLight example to exchange the emulated traffic sensor data between the threads.

The control block for the message queue object is allocated via object-specific memory in RTX and **Number of Message Queue objects** is set to 1.

Memory for data storage is allocated by the system and so will be mapped to privileged RAM area so that it can be accessed from non-privileged threads only via RTOS APIs. Data storage size is set to 128 bytes.

See section **Communication across safety classes** for more details on how the message queue is used.

Event Recorder configuration

EventRecorder is enabled in the TrafficLight example as explained in **Usage of FuSa Event Recorder**.

RTX_Config.h contains some global configuration options for the Event Recorder operation, as well as specific to the RTOS events.

Global initialization is enabled in the Event Configuration group as FuSa Event Recorder will be initialized by RTX. **Start recording** flag is set which triggers event recording right after initialization.

The events generated by the RTOS are set in **RTOS Event Generation Group** and can be modified to enable/disable specific event types.

Additionally, the **Compiler** component contains the file *EventRecorderConf.h* that configures the size of the Event Recorder buffer as well as the time stamp source and frequency.

Other RTX configuration options

Mutexes are used by the MDK-Middleware Network component, but are statically allocated, so no special configurations are required in the *RTX_Config.h* file.

Semaphore and Memory pool objects are not used in the TrafficLight example and corresponding configurations are not relevant for it.

System initialization

On Cortex-M devices, the execution starts in privileged mode after reset. Also, the MPU is disabled, so all memory can be accessed without any restrictions.

The *Reset_Handler* exception is implemented in the device startup file (*startup_stm32f407.s, startup_stm32f746xx.s*). From there, the *SystemInit(void)* performs initial device configuration and then the pre-main function *__main(void)* is called.

The pre-main function *__main()* needs to be implemented in the application as it is not available in the FuSa C library. FuSa C lib safety manual explains the expected startup sequence and in the TrafficLight example it is implemented in *fusa_clib_startup.c.* At the end, the application *main()* function is called.

In the *main.c* , the *main()* function initializes HAL and Clocks, as well as peripherals for LED and ADC.
A hardware watchdog is initialized and started with 1 second timeout to act as the last-resort protection against system hang.

MPU initialization

ARM_MPU_Enable(MPU_CTRL_PRIVDEFENA_Msk); is called to enable MPU with access to the default memory map from the privileged mode. Such access is required for correct kernel operation, but also enables access to the memory from all interrupts.

RTOS initialization

The FuSa RTX kernel is set up with *osKernelInitialize().*

osKernelProtect() is called to disable kernel control from threads lower than a specific safety class. In our example, *SAFETY_CLASS_COMMUNICATION* is used in the argument, thus disabling kernel control from the normal operation mode (threads have lower safety class value). This is done because in our example, from communication threads we want to trigger some faults by suspending the kernel. This is needed only for demo purposes and in a real application, only threads with a high safety class that need to control the kernel should be able to do so.

Note that in cases when device needs to go into sleep, this would typically be done in the Idle thread. Hence the idle thread would need to have a sufficient safety class and potentially privileged level for suspending the kernel and putting the system into sleep. In our example we do not put the device into low-power mode.

With *osKernelStart()* the scheduling process starts, so it is called after initial application threads are created.

Thread initialization

User threads are created for the application operation modes. The safety class and MPU Protected Zone values are assigned when RTX objects are created.

The following code snippet shows how the *SensorThread* is created in the *NormalOperation.c* file. The local array *sensor_thread_stack* is allocated for the thread stack.

```
/* Threads stacks located in RAM_NORMAL_OP */
static uint64_t sensor_thread_stack [512/8];

/* SensorThread thread attributes */
static const osThreadAttr_t sensor_thread_attr = {
  .name       = "SensorThread ",
  .attr_bits  = osThreadUnprivileged                        |
                osThreadZone(ZONE_NORMAL_OP)                |
                osSafetyClass(SAFETY_CLASS_NORMAL_OPERATION),
  .cb_mem     = NULL,                             /* System allocated control block */
  .cb_size    = 0U,
  .stack_mem  = sensor_thread_stack,              /* User provided stack */
  .stack_size = sizeof(sensor_thread_stack),
  .priority   = osPriorityNormal,
  .tz_module  = 0U,                               /* Not used */
  .reserved   = 0U
};

(void)osThreadNew(SensorThread, NULL, &sensor_thread_attr);
```

SensorThread is created with unprivileged access level (*osThreadUnprivileged* in *attr_bits*). Its MPU Protected Zone is set to *ZONE_NORMAL_OP* (0) and safety class to *SAFETY_CLASS_NORMAL_OPERATION* (0).

Variables defined in *NormalOperation.o,* including *sensor_thread_stack*, get placed by default into the *RAM_NORMAL_OP* area as defined in *scatter.sct* file:

```
RAM_NORMAL_OP REGION_RAM_NORMAL_OP_START REGION_RAM_NORMAL_OP_SIZE {
  NormalOperation.o (+RW +ZI)
}
```

and this memory region is also defined in the MPU Protected Zone number 0 in *zones.c* file that the *SensorThread* is assigned to.

Notes:

- Shared variables defined in *NormalOperation.c* are explicitly placed into shared RAM as explained in section **Usage of shared resources**.
- Variables defined as *static const* (for example *sensor_thread_attr*) are placed into flash memory with read-only access.

Section **MPU Protected Zones** provides additional details on how MPU Protected Zones are defined and used.

In the example, *StlThread* is created for execution in privileged mode as this is required by X-CUBE-STL. All other threads are created to run in unprivileged mode.

After the application threads are created, the function *osThreadProtectPrivileged()* is called to disable the creation of new threads with privileged access level.

RTX kernel operation

FuSa RTX RTOS kernel operations are mostly executed in handler mode using exception handlers *SysTick_Handler, SVC_Handler* and *PendSV_Handler*.

Exceptions and interrupts are executed in privileged handler mode and from there any memory can be accessed since during **System initialization** MPU is enabled with *MPU_CTRL_PRIVDEFENA_Msk.*

In the *scatter.sct* file, the RAM used only by RTOS is explicitly mapped into *RAM_PRIVILEGED* area. This ensures that kernel data does not accidentally end up in regions accessible from threads.

```
RAM_PRIVILEGED REGION_RAM_PRIVILEGED_START REGION_RAM_PRIVILEGED_SIZE {
  Net_Config.o (.bss.os.*.cb)
  * (.data.os*)
  * (.bss.os*)
}
```

The control blocks of RTOS objects need to be in this memory.

- Section *Net_Config.o (bss.os*.cb)* places the control blocks for the objects defined in the Network MDK-Middleware component into the privileged RAM. See also **Usage of the MDK-Middleware network** component.
- ** (.data.os*)* corresponds to the object-specific memory buffers used by the RTOS.
- *(.bss.os*)* is used for kernel internal items, as well as object control blocks. When static memory allocation is used in the application, the control blocks for the objects will need to be placed into corresponding part of .bss.os. The section **Usage of the MDK-Middleware network** component shows how this is done in case of networking threads.

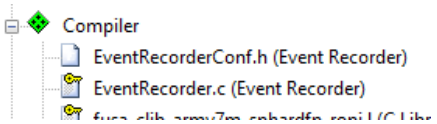
Additionally, the main stack used in the exceptions is placed into a special memory region that will not be accessible from threads and is kept uninitialized to ensure that the program starts correctly after pre-main function *__main* is executed.

```
ARM_LIB_STACK REGION_ARM_LIB_STACK_START EMPTY REGION_ARM_LIB_STACK_SIZE {
}
```

The section **FuSa RTX configuration** explains RTX configuration options for process isolation and describes how internal threads operated by RTX kernel (Idle thread and Timer thread) are configured.

Usage of FuSa Event Recorder

In the Traffic Light example, the Event Recorder (FuSa variant) is selected for the **Debug** target in the **Manage Run-Time Environment** window, and corresponding files can be found in the Project window in the **Compiler** component.



Event Recorder is disabled for the **Release** target. Alternatively, it is also safe to keep the Event Recorder in the **Release** target, and if needed, just disable the event generation if it is not required for logging purposes.

Event Recorder does not use separate threads and its API is called from different components, including the RTX kernel. So, it cannot be classified with a single safety class, or allocated only to certain MPU Protected Zone. Event Recorder RAM is placed into a dedicated *RAM_EVR* area that is then accessible for all MPU Zones (refer to the *scatter.sct* file):

```
RAM_EVR REGION_RAM_EVR_START UNINIT REGION_RAM_EVR_SIZE {
   EventRecorder.o (+ZI)
}
```

The region *RAM_EVR* is defined as uninitialized as required by FuSa Event Recorder.

Usage of the MDK-Middleware network component

For communication, the TrafficLight example uses the **MDK-Middleware Network** stack. The application implements a simple HTTP-server.

The file *Communication.c* implements the application thread *CommunicationInitThread* that first initializes the underlying MDK-Middleware Network with the *netInitialize()* function, and then constantly verifies if a memory access fault was triggered from the web interface in order to inject it into the system (see section **Web interface**).

The *CommunicationInitThread* is defined as part of an object-specific memory block so its control block is automatically placed into the correct memory region for privileged access by the RTOS. The thread stack (*communication_init_thread_stack*) gets mapped into *RAM_COM* area in the *scatter.sct* file.

The thread stacks as well as other data items that are necessary for communication are placed into an MPU region dedicated to the communication *RAM_COM*. Below is an example for NUCLEO-F429ZI:

```
RAM_COM REGION_RAM_COM_START REGION_RAM_COM_SIZE {
    Communication.o (+RW +ZI)
    *Net*.lib (+RW +ZI)
    Net_Config.o (.bss.*)
    emac_stm32f4xx.o (+RW +ZI)
    phy_lan8742a.o (+RW +ZI)
  }
```

Additionally, functions for sharing or modifying the data from normal operation mode are implemented. Corresponding data items such as *u32_input_val*, *u32_output_val*, and *str_fault_info* are placed into the shared RAM region (see **Communication across safety classes**).

In the TrafficLight example, two threads are created by the network stack library: *netCore_Thread* and *netETH0_Thread* (in *net_rtos2.h*). The control blocks for these threads as well as related RTOS objects already get placed into the corresponding named section *.bss.os* in the *Net_Config.o* object. This can be seen in *net_rtos2.h* file.

As described in **RTX kernel operation** these *.bss.os* sections need to be accessible from the RTOS kernel only, and hence are mapped in the linker scatter file as part of the *RAM_PRIVILEGED* area.

User SVC calls

The Ethernet operation in TrafficLight example requires some control over the Ethernet interrupts. Specifically, it needs to be able to execute three NVIC functions: *NVIC_EnableIRQ(IRQn)*, *NVIC_DisableIRQ(IRQn)* and *NVIC_ClearPendingIRQ(IRQn).*

However, the networking threads are executed in unprivileged mode and hence cannot directly modify necessary MCU registers because that would trigger an exception. To overcome such a restriction, it is possible to wrap required code in user SVC (SuperVisor Call) calls that implement the required functionality.

In our implementation, we use the *CMSIS_NVIC_VIRTUAL* define that allows to override NVIC functions which are defined in CMSIS-Core and are mapped to and implemented in the *svc_user.c* file.

In the application, the file *cmsis_nvic_virtual.h* already maps NVIC functions calls to their implementations in CMSIS-Core (*__NVIC_...*).

In the TrafficLight example, such interrupt control functionality is needed only for Ethernet interrupts, so there is a dedicated verification that the call is done for Ethernet IRQs and from the communication MPU Zone.

```
/* SVC handler for NVIC functions:
   - NVIC_EnableIRQ
   - NVIC_DisableIRQ
   - NVIC_ClearPendingIRQ
   it enables these functions to be executed in privileged mode */
void svcNVIC_Handler (uint32_t func_index, IRQn_Type IRQn) {

  /* Only Ethernet IRQ handling from Zone 2 is allowed */
  if (IRQn == ETH_IRQn) {
    if (osThreadGetZone(osThreadGetId()) == ZONE_COM) {
      switch (func_index) {
       …
      }
    }
  }
}
```

This approach can also be used for access to generic peripherals as explained in section **Peripheral**.

MPU Protected Zones

The table provided in **Application overview** lists the MPU Protected Zones assigned to specific threads and the memory regions enabled in them. This section explains the implementation details.

Zone definitions

MPU Zones are defined in the files located in the *./CMSIS_Zone/ftl_gen/* folder.

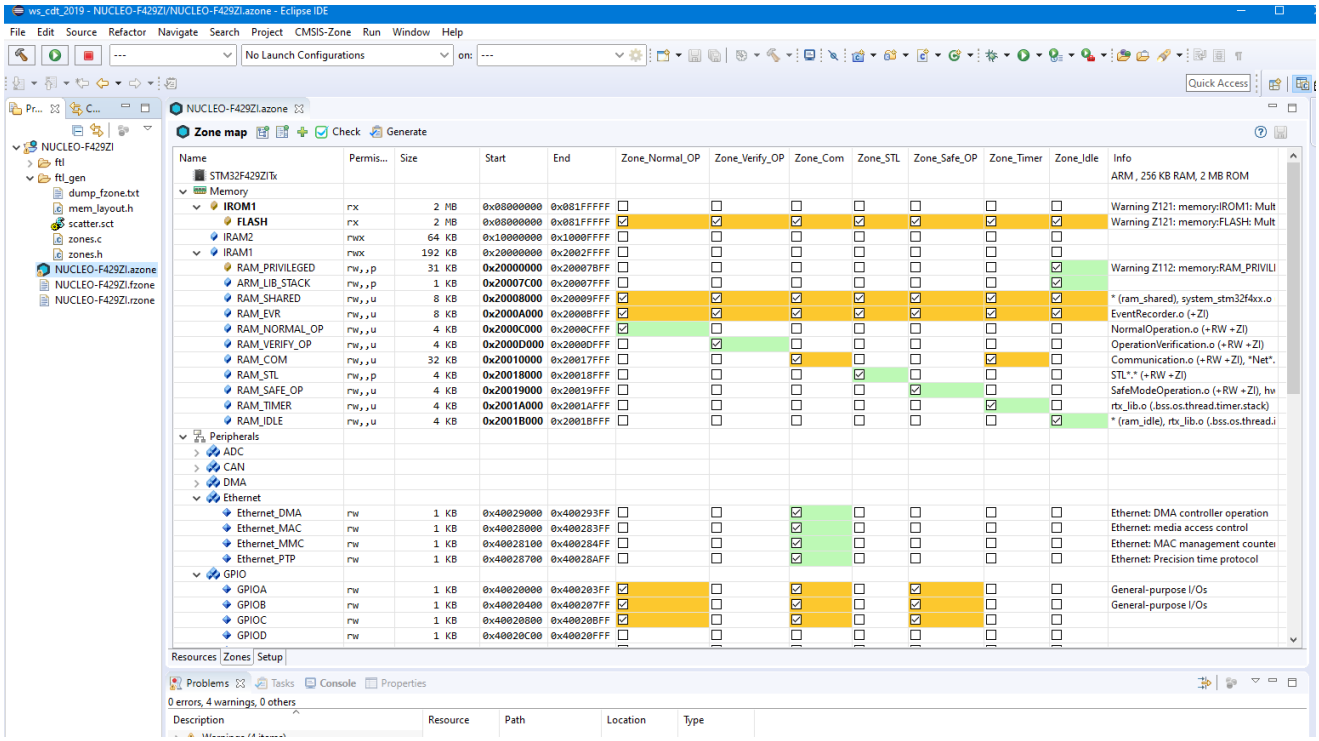
- *zones.h* and *zones.c* files define the MPU Table of type *ARM_MPU_Region_t* (*mpu_table).* Each element in the array defines an MPU Protected Zone, with each row specifying a memory region and its access rights. In total 7 zones (*ZONES_NUM*) are defined and they are referenced in the **Application overview.**
- *mem_layout.h* contains defines for addresses of memory regions used for memory organization by the scatter file.
- *scatter.sct* is a scatter-loading file that allocates memory regions as defined in the *mem_layout.h* and places the application object files into the specific memory regions.

These files can be maintained manually, but also generated using graphical **CMSIS-Zone utility.**

The directory *./CMSIS_Zone/* contains the project files that are used in the CMSIS-Zone utility:

- *.rzone* – generic file that describes resources available on the target microcontroller.
- *.azone* – project-specific file that describes MPU Zones allocations, memory layout and mapping of application object files.
- *.fzone* – is autogenerated by CMSIS-Zone utility for internal use and shall not be modified or used otherwise.

This is a screenshot of the CMSIS-Zone utility for the NUCLEO-F429ZI target project:



Techniques for memory mapping

The TrafficLight example shows some useful techniques that help to place individual program objects into target memory regions. The **Arm Compiler 6.6 documentation** describes available methods in more details.

1) **Map all global data defined in a C file**
   This is the simplest approach. Data items can be implemented in separate C files and then the corresponding object files can be placed to the target memory region using scatter file. In our example this is done for EventRecorder:

```
RW_RAM_EVR REGION_RAM_EVR_START UNINIT REGION_RAM_EVR_SIZE {
   EventRecorder.o (+ZI)
}
```

2) **Map an application data item into a named section**
   If only some data defined in a C file needs to be placed in a specific memory region, then `__attribute__` can be used to place it into a named memory region. This name can be then mapped to a corresponding memory section defined in the scatter file. For example, in the *NormalOperation.c* file, *sensor_data_mq* is defined in the *ram_shared* section:

```
/* Local variable located in shared RAM */
static osMessageQueueId_t sensor_data_mq __attribute__((section("ram_shared"))) = NULL;
```

   and the *ram_shared* section is mapped into *RAM_SHARED* in the *scatter.sct* file (along with system and HAL) as described in **Usage of shared resources**.

3) **Map a data item without modifying the code (for MDK-Middleware or RTOS)**
   In some cases, it is not desired or not possible to modify a variable definition. It is possible to map such data to a target memory region. For example, the following definition places the stack of the Timer thread into the dedicated RAM region:

```
RAM_TIMER REGION_RAM_TIMER_START REGION_RAM_TIMER_SIZE {
    rtx_lib.o (.bss.os.thread.timer.stack)}
```

In this case of the timer thread stack, it is already being placed into a specific named memory *.bss.os.thread.timer.stack* and that is why in the scatter file we just map it to the RAM_TIMER region. Refer to the *rtx_lib.c* file where the timer thread stack is defined as follows:

```
// Timer Thread Stack
static uint64_t os_timer_thread_stack[OS_TIMER_THREAD_STACK_SIZE/8] \
__attribute__((section(".bss.os.thread.timer.stack")));
```

Usage of shared resources

Items that need to be accessed from different MPU Protected Zones should be placed into a RAM region that is then specified as accessible by those Zones.

In our example, two RAM regions are read/write accessible for all MPU Zones.

1) The memory region *RAM_SHARED* is defined in the *scatter.sct* file:

```
RAM_SHARED REGION_RAM_SHARED_START REGION_RAM_SHARED_SIZE {
  * (ram_shared)
  system_stm32f4xx.o (+RW +ZI)
  stm32f4xx_hal.o (+RW +ZI)
}
```

It contains following items:

- STM32F4 HAL (*stm32f4xx_hal.o*).
- System device file (*system_stm32f4xx.o*).
    - *SystemCoreClock* variable defined in system file is required by *HAL_GetTick* function that is frequently called from HAL.

  Note that in this example complete HAL and system device data are placed for shared access and can be overwritten also by components with lower safety integrity levels. This is done so for simplicity reasons only. In a real-world application, the potential safety impact needs to be analyzed and potentially additional provisions are needed to protect such data. Also see **Peripheral**.
- *ram_shared* is a named section used in the example for individual items that are intended for access from various MPU Protected Zones.
    - HAL ticks defined in *HAL_GetTick* function in *main.c.*
    - Variables that are shared with communication threads for display on the web page
        - Sensor input, light status, and operation mode
        - Fault information array
    - Id of the event flag object used for triggering faults
    - Id of the event flag object used for starting safe operation mode
    - Id of the message queue object used for sharing traffic data

  Note: To be able to access and modify an RTOS object from threads assigned to different MPU Protected Zones, at least the following is required:

    - The object Id needs to be placed into the shared RAM
    - The assignment of safety class to the object and the threads shall enable required access. Most manipulations with RTOS objects are permitted only for threads that have the same or higher safety class as assigned to the target RTOS object.

  Also see **Communication across safety classes**.

2) The memory region *RAM_EVR* is allocated for the Event Recorder buffer, so that all threads can record events. See section **Usage of FuSa Event Recorder** for details.

Peripheral access protection

For correct operation, peripheral drivers require read/write access to various CPU peripheral registers. MPU Protected Zones can define corresponding access rights to the memory-mapped peripherals. Often, the target registers are not peripheral-specific and define system-wide behavior.

In the TrafficLight project for example, the Ethernet MAC driver (implemented in *EMAC_STM32F4xx.c* and called from the thread *netETH0_Thread* (part of MPU Zone 2)) requires not only access to the Ethernet MAC specific peripheral (*ETH*) but also to generic System Configuration (*SYSCFG*), Reset and Clock Control (*RCC*) as well External Interrupts (*EXTI*) register groups.

If such a generic peripheral is enabled in a low safety integrity MPU Protected Zone, then any code from that Zone has access to the peripheral. This implies a risk that the low safety code can modify the parts of the peripheral not required for the operation of a specific driver (Ethernet in our example) and thus impact the safety-critical components that depend on the same peripheral. Such safety risks must be analyzed and if necessary mitigated. Peripheral drivers used in the current version of the TrafficLight example do not address this problem.

One potential approach could be to use aa SVC gateway. In this approach, the low safety MPU Protected Zone does not enable the peripheral for access and hence the non-privileged code cannot unintentionally modify the peripheral. The access from the privileged mode is enabled through the background region (in the TrafficLight example done with *ARM_MPU_Enable(MPU_CTRL_PRIVDEFENA_Msk);*). The peripheral driver needs to rely on user SVC functions that are executed in privileged mode but also should implement explicit control that only specific peripherals or parts of them are accessed. NVIC control functions explained in section **User SVC calls** demonstrate how this approach can be implemented.

Note: When running in the privileged mode, all other peripherals and memory regions can be accessed as well. Hence implementation of user SVC calls must be extensively analyzed and validated.

Zone assignments to threads

Individual threads are assigned to the MPU Zones when the threads are created. Section **Thread initialization** provides an example.

Loading zones

An MPU Zone is loaded in the *osZoneSetup_callback(uint32_t zone)* function. It is called by the RTX kernel during a thread switch if the next thread to be executed has a different MPU Protected Zone than the currently one. Implementation is done in the *zone.c* file and is quite straightforward:

```
/* Update MPU settings for newly activating Zone */
void osZoneSetup_Callback (uint32_t zone) {

  if (zone >= ZONES_NUM) {
    ZoneError_Handler();
  }
  ARM_MPU_Disable();
  ARM_MPU_Load(mpu_table [zone], MPU_REGIONS);
  ARM_MPU_Enable(MPU_CTRL_PRIVDEFENA_Msk);
}
```

Handling memory access faults

When a memory access violates the rules of the currently loaded MPU Zone then a Memory Management Interrupt (MemoryManagement_IRQn) is triggered by the processor and its handling function is executed according to the exception vector table specified in the device startup file (by default, *MemManage_Handler*).

The example application provides an implementation of the *MemManage_Handler()* in the *faults.c* file. It is implemented in assembler to avoid overwriting the link register. From there the *MemManage_Handler_in_C()* function is called that implements the actual handling as summarized in the table below:

Condition	Actions
The fault is triggered from exception context	Trigger fatal error, see *FatalError_Handler* in **Fault handling**
The fault is caused from an invalid thread (thread Id is NULL, or MPU protected Zone is invalid)	
The fault is caused by safe mode operation	
The fault is caused from a non-safe operation. Can be injected from web interface.	Suspend operation of the threads assigned to the MPU Protected Zone that has caused the failure. Log fault info and send it to the web server.

In the last case, when the fault is caused from a non-safe operation, the execution continues and at the end of the *MemManage_Handler* the *osFaultResume* function is called to correctly resume RTOS operation.

The section **Fault handling** additionally explains how other faults and errors are handled in the TrafficLight example.

Safety classes

The example uses various RTOS objects such as threads, event flags, timer, mutexes and message queue. They are defined in the application code as well as in the MDK-Middleware Network library.

Assignment of safety classes is typically done first for thread objects based on the safety integrity level they implement. For TrafficLight, this is explained in subsection **Safety class assignment to threads**.

An RTOS object can use a default safety class (0) or inherit the safety class of a thread that creates it if the object will be used only by threads with the same safety class. In the TrafficLight example, this is the case for the threads and objects created by the MDK-Middleware Network library. They all operate with the SAFETY_CLASS_COMMUNICATION value inherited from the *CommunicationInitThread*. The section **Usage of the MDK-Middleware network** component explains the integration.

RTOS objects that shall be used by threads with different safety classes require additional considerations as explained in section **Communication across safety classes**.

Safety class assignment to threads

The table provided in **Application overview** lists the safety classes assigned to specific threads and operation modes. File *system_defs.h* contains safety class value definitions assigned to specific numeric values and they are used throughout the application.

The section **Thread initialization** shows a code snippet that creates the *SensorThread* with safety class *SAFETY_CLASS_NORMAL_OPERATION* (0) assigned to it.

Communication across safety classes

Different mechanisms can be used to exchange data between threads of different safety classes.

1. **Use variables placed in a shared RAM region.**

   Section **Usage of shared resources** explains how shared access to a memory region can be enabled using MPU Protected Zones.

   For example, the *Communication.c* file defines variables that are used for displaying system information. Set/get functions implement the access interface that can be called by various threads.

   The variables are mapped to a named section *ram_shared.* This section is part of the memory region RAM_SHARED that is enabled for access in all MPU Protected Zones defined in the example.

   For example, the *sensor_val* variable is accessible via the functions *DisplaySetSensorValue(..)* and *DisplayGetSensorValue(..)*. The variable is defined in *Communication.c* as follows:

   ```
   /* Local variables located in shared RAM */
   static uint32_t   sensor_val    __attribute__((section("ram_shared"))) = 0U;
   ```

   Note that in this approach the data exchange is not performed via the RTOS but is fully under the control of the application.

2. **Use RTOS objects.**

   In the TrafficLight example, following RTOS objects are used by threads with different safety classes and different MPU Protected Zones and are placed in shared RAM area as explained in **Usage of shared resources**:

   - **Event flag object *fault_inject_event*.**

     It is defined in *Communication.c* and is used for triggering faults in the program.

     Here are the key specifics of its implementation that make it accessible from threads with different safety classes and MPU Protected Zones:

     - The object Id *fault_inject_event* is mapped to a shared RAM region accessible from MPU Protected Zones assigned to the target threads. See **Usage of shared resources** for further details.
     - The object is explicitly assigned with the lowest safety class value (0) and so can be accessed from all threads. The FuSa RTS Safety manual **[6]** lists RTOS API functions where safety class assignment gets verified. If the target object has a higher safety class than the safety class of the running thread, then the requested operation will be rejected.
     - The object control block is allocated by kernel. Also see **Event flags configuration**. In case of static memory allocation, the application should place the control block to privileged RAM area as mentioned in **RTX kernel operation** and required in FuSa RTS Safety manual **[6]**. This requirement is generic for RTOS objects and is independent on their use by different safety classes.
     - In the example, the functions *FaultInject* and *FaultWasInjected* access the object. They can be called by all threads because the TrafficLight application does not partition flash code with MPU Protected Zones. Access protection for the data (object Id) and safety class verification is sufficient in this case.

This code snippet shows how *fault_inject_event* is created:

```
/* Local variable located in shared RAM */
static osEventFlagsId_t fault_inject_event __attribute__((section("ram_shared"))) = 0U;

/* fault_inject_event event flags attributes */
static const osEventFlagsAttr_t fault_inject_event_attr = {
  .name       = "fault_inject_event",
  .attr_bits  = osSafetyClass(0U),
  .cb_mem     = NULL,                          /* System allocated control block */
  .cb_size    = 0U
};
…
void CommunicationInit (void) {
  fault_inject_event = osEventFlagsNew(&fault_inject_event_attr);
   …
}
```

- **Event flag object *safe_mode_operation_event_id.***

  Defined in *SafeModeOperation.c;* is used for starting safe operation mode. The implementation concept equals the one of *fault_inject_event* explained above.

- **Message queue object *sensor_data_mq*.**

  Defined in *NormalOperation.c* and is used for sharing sensor data that emulates traffic density. Implementation concept equals the one of *fault_inject_event* explained above.

  - Note that the message queue is created with the memory for data storage being allocated by the kernel as explained in **Message queue configuration**. This prohibits direct access to the message queue data memory from non-privileged threads. Access is allowed only using message queue RTOS APIs with *sensor_data_mq* object Id value that is in the shared RAM area.

Thread watchdogs

The example demonstrates how thread watchdogs can be used in an application. The table below lists the threads and the thread watchdogs maintained for them:

Thread	Operation mode	Thread Watchdog interval, ms
NormalOperationThread	Normal operation	200
SensorThread	Normal operation	Not used
OperationVerificationThread	Operation verification	100
StlThread	STL	1500
SafeModeOperationThread	Safe operation	500
CommunicationInitThread, netCore_Thread, netETH0_Thread	Communication	Not used
osRtxTimerThread	Timer thread	Not applicable
osRtxIdleThread	Idle thread	Not used

Thread watchdogs are typically restarted at the beginning of the infinite loop (*for(;;)* or *while(1)*) present in the thread. For communication threads, there is no thread watchdog as the example has no real-time requirements.

Thread watchdog alarm handler

If a thread watchdog does not get fed within the expected time interval, the *osWatchdogAlarm_Handler* callback function is called (by the RTOS) with the thread Id as the parameter indicating for which thread the watchdog has expired. Based on that, the application can tailor its response. The function should return a new value for the thread watchdog, or 0 if the thread watchdog should not be restarted.

In the example, the thread *osWatchdogAlarm_Handler* is implemented in *faults.c* file.:

- Initial checks verify the thread ID parameter and the safety class assigned to it.
- If the thread watchdog is triggered for the *ThreadSafeModeOperation* thread, then *osThreadSuspendClass* is called to suspend all threads with lower safety class. This helps to free the kernel resources for the safe mode operation so that its execution can be scheduled. The new value for the thread watchdog is set to 510 ms. However, within the *ThreadSafeModeOperation* also the hardware watchdog is fed, so now if this thread will not be served in 500ms then the hardware watchdog will trigger and reset the system.
- If the thread watchdog expires for a thread other than *ThreadSafeModeOperation*, then we activate the safe mode operation (switching the traffic lights to blinking yellow mode) and suspend all threads of the same or lower safety class.

Note that if a thread gets suspended from scheduling while the application calls such functions as *osThreadSuspend* or *osThreadSuspendClass*, its thread watchdog continues to run, and it is expected to expire and trigger the alarm function as the thread will not be running anymore.

Hence, it may be important to differentiate the handling of a thread watchdog that expired unexpectedly, from one that gets fired when the application intentionally suspends the operation of a specific thread.

Fault handling

Arm FuSa RTS implements mechanisms that can detect certain failures and allows suspending parts of the system from operation. This can be used when handling faults and errors to reduce the current scope of the application operation and so avoid inference from the code with lower safety-criticality level.

In the example, *faults.h*/*faults.c* implement the error and fault handlers. The next table summarizes them:

Fault/Error	Actions
NormalOperationError_Handler Called from *OperationVerificationThread* if input data about measured traffic is not plausible. Executed in thread context. Can be injected by potentiometer on the board.	Activate safe mode operation. Suspend threads with same or lower safety class. Log error message and send to the web server.
StlError_Handler Called when STL detects a failure or fails to run. Executed in thread context. Can be injected from web interface.	Activate safe mode operation. Suspend threads with safety class lower than safe mode thread. Log error message and info to the web server.
osWatchdogAlarm_Handler Callback issued by RTX kernel if a thread watchdog expires. Executed in SVC context. Can be injected from web interface.	See **Thread watchdog alarm handler**.
SystemStartupError_Handler Clock setup fails.	Trigger fatal error, see *FatalError_Handler*.
MemManage_Handler Called when memory access has violated the rules configured for the MPU Protected Zones. Executed in SVC context. Can be injected from web interface.	See **Handling memory access faults**.
SafeModeDormantError_Handler While in normal operation mode, an error occurred with the event flag that signals safe mode operation (for example event flag was destroyed)	Activate safe mode operation. Suspend threads with safety class lower than safe mode thread. Log error message and send to the web server.
ZoneError_Handler An undefined zone is requested for activation in *osZoneSetup_Callback*.	Trigger fatal error, see *FatalError_Handler*.
osRtxErrorNotify Notifies that RTX has detected an issue during its operation, such as thread stack underflow, ISR queue overflow, and others. Executed in SVC context.	Action depends on the reported error. *osRtxErrorStackUnderflow* • Fatal error when in safe operation, or in unknown state (see *FatalError_Handler)* • Otherwise activate safety mode, suspend threads with same or lower safety class. *osRtxErrorISRQueueOverflow* • Fatal error. See *FatalError_Handler.* *osRtxErrorTimerQueueOverflow* • Activate safety mode, destroy all objects with safety class lower than safe mode. Other: • Fatal error. See *FatalError_Handler.*
FatalError_Handler No recovery path possible/defined.	Turn on red light and reset the system.

Injecting faults

The application provides a **Web interface** that can be used to control and monitor program execution.

- **Data error:**
  - If emulated traffic data is not plausible (larger than 4000 or timed-out), this is considered by the application as a data error that requires transition to a safe state.
    To trigger the data error, doble-click the USER button on a NUCLEO board or turn the potentiometer to maximum on the MCB board. This results in the following operation:
    - *NormalOperationError_Handler* is called. It starts safe mode operation and suspends all threads with the same or lower safety class (0).
    - Web interface and communication continue working and display safe mode operation.
    - The STL thread continues its operation.
    - The LED status indicates safe operation mode.
    - *OperationVerificationThread* was suspended, so its thread watchdog expires.

      This gets captured in the fault info. In this case, handling of the thread watchdog alarm does not change the operation as the application is already in the safe mode and threads with safety class 0 are suspended.
    - *NormalOperationThread* was suspended, so its thread watchdog expires.

      This gets captured in the fault info. In this case, handling of the thread watchdog alarm does not change the operation as the application is already in the safe mode and threads with safety class 0 are suspended.
    - As the result, the **Fault info** field gets extended with following error messages:

      ```
      Operation Verification has detected an error in input data
      Thread watchdog alarm was triggered for thread OperationVerificationThread
      Thread watchdog alarm was triggered for thread NormalVerificationThread
      ```

The **Inject fault** section contains buttons that can be used to inject different faults into the system.

- **Thread watchdog:** locks the specified thread to trigger its **Thread watchdog alarm handler.**
  - **OperationVerificationThread** button:

    This works in the Normal mode only. In the Safe mode no action is performed.
    - Thread watchdog alarm handler is triggered for *OperationVerificationThread*. It starts safe mode operation and suspends all threads with the same or lower safety class (0).
    - Web interface and communication continue working and display safe mode operation.
    - The STL thread continues its operation.
    - The LED status indicates safe operation mode.
    - Since *NormalOperationThread* was suspended its thread watchdog expires as well.

      This gets captured in the fault info. In this case, handling of the thread watchdog alarm does not change the operation as the application is already in the safe mode and threads and threads with safety class 0 are suspended.
    - As the result, the **Fault info** field gets extended with following error messages:

      ```
      Thread watchdog alarm was triggered for thread OperationVerificationThread
      Thread watchdog alarm was triggered for thread NormalVerificationThread
      ```

- **SafeModeOperationThread** button:
  - The thread watchdog alarm handler is triggered for *SafeModeOperationThread*. It suspends all threads with lower safety class and initiates hardware watchdog to verify the recovery.
  - Since the lock is injected into the *SafeOperationThread* itself, the attempt to recover by suspending other threads does not help.
  - Initiated hardware watchdog expires, and the system is reset.
  - System starts in normal operation mode, **Fault info** displays:

    ```
    The system was reset by a hardware watchdog
    ```

- **Memory access:** performs memory access outside of the MPU Protected Zone assigned to corresponding thread to trigger memory access fault (*MemManage_Handler*)
  - **NormalOperationThread** button:

    Works in the Normal mode only. In the Safe mode no action is performed.
    - Upon invalid memory access the *MemManage_Handler* handler terminates *NormalOperationThread.*
    - As the result, an error in normal operation mode is detected (*NormalOperationError_Handler*) that transitions system to a safe mode and suspends user threads with the same or lower safety class (0).
    - Web interface and communication continue working and safe mode operation is shown.
    - Since *OperationVerificationThread* is suspended now, its thread watchdog expires, and corresponding alarm handler is triggered.
    - The STL thread continues its operation.
    - The LED status indicates safe operation mode.
    - As a result, the **Fault info** field gets extended with following error messages:

      ```
      Memory fault caused by NormalOperationThread trying to access data at address 0x20000000
      Operation Verification has detected an error in input data
      Thread watchdog alarm was triggered for thread OperationVerificationThread
      ```

  - **CommunicationInitThread** button:
    - Upon invalid memory access, the *MemManage_Handler* handler terminates all communication threads, but the system stays in the current operation mode.
    - The STL continues to execute.
    - The LED status indicates the current operation mode.
    - Web interface is not responding as communication threads are terminated.

- **STL:** injects a CPU fault (handled in *StlError_Handler*) as if it was detected by an STL test. Works only if the STL is enabled in the project, otherwise no reaction.
  - **CPU fault** button:
    - The STL diagnostic test detects a CPU fault.
    - *StlError_Handler* transitions system to a safe mode and suspends user threads with the same or lower safety class as the STL thread (2).

        - Since *OperationVerificationThread* is suspended now, its thread watchdog expires, and the corresponding alarm handler is triggered.
        - Since *NormalVerificationThread* is suspended now, its thread watchdog expires, and the corresponding alarm handler is triggered.
        - The LED status indicates safe operation mode.
        - Web interface is not responding as communication threads are suspended.
- **Fatal:** suspends the RTOS kernel by calling *osKernelSuspend()*.
    - **Suspend kernel** button:
        - This triggers a chain of various errors that could not be recovered.
        - Initiated hardware watchdog expires, and the system is reset.
        - System starts in normal operation mode and **Fault info** displays:

```
The system was reset by a hardware watchdog
```

Summary

In this application note, we have shown how to use process isolation capabilities provided in FuSa RTS to implement a complex example with processes of different safety integrity levels. The example projects are provided for selected development boards with Cortex-M4 and Cortex-M7 devices, but the concepts can be applied on any Cortex-M device with MPU as supported by FuSa RTS.

References and useful links

[1] Arm FuSa RTS
[2] µVision User's Guide
[3] ULINKpro User's Guide
[4] Arm Safety Compiler
[5] AppNote 326: Using X-CUBE-STL with Arm FuSa RTS
[6] Arm FuSa RTS Safety manual