| Document Title | Explanation of Safe API for hardware accelerators |
|---|---|
| **Document Owner** | AUTOSAR |
| **Document Responsibility** | AUTOSAR |
| **Document Identification No** | 1086 |

| | |
|---|---|
| **Document Status** | published |
| **Part of AUTOSAR Standard** | Adaptive Platform |
| **Part of Standard Release** | R23-11 |

| Document Change History | | | |
|---|---|---|---|
| **Date** | **Release** | **Changed by** | **Description** |
| 2023-11-23 | R23-11 | AUTOSAR Release Management | • Initial release |

**Disclaimer**

# Contents

# 1 Introduction

This document summarizes the motivation, objective and make recommendations for usage of "Safe API for hardware accelerators". A short name can be used: "Safe HWA API". Strategic goal is to make Safe HWA API as built-in functionality into AUTOSAR Adaptive platform.

Safe HWA API is a concept which provides possibility to utilize platform available hardware accelerators for high performance computing independently of hardware presence physically or in a virtualized form (e.g., hypervisor virtualization)

**NOTE**

Safe HWA API concept is only defining the API and providing requirements for it. The actual Safe HWA API implementation and integration of all needed libraries will be the responsibility of the AP Platform Vendor.

## 1.1 Objectives

The main goal of Safe HWA API is to enable parallel heterogeneous programming (general information can be found here [1]) in a convenient and efficient way using a standardized C++ based API. Mentioned can leverage diverse processors to utilize all available hardware for solving the issue of high-performance computing. Also, the current concept was developed with respect to ISO-26262, part 6, which will increase reliability of Automotive software developed for high performance computing. The safety for hardware, vendor's implementation and underlying technology libraries are also highly demanded but is out of scope of the current concept. Especially parallel heterogeneous programming will be actual for AD/ADAS development domain, which is a strategic goal for AUTOSAR Adaptive.

Safe HWA API will allow application developers to delegate the computation of specific tasks to the most suitable hardware (e.g. GPU, FPGA etc.). There are two options available:

- Choose particular hardware accelerator for task execution in runtime
- Use precompiled kernel function for specific hardware accelerator

Also, there are convenient methods for tasks ordering and synchronization. The choice of particular backend implementation, which supports different hardware, will not affect client code.

## 1.2 Scope

Scope of current document is to introduce Safe HWA API, describe main approaches to use it efficiently by application developers.

In this document we will go through Safe HWA API architecture and explain main functionality. We will examine coding examples that provide the reader with a deeper understanding of how to use Safe HWA API in practice.

# 2 Definition of terms and acronyms

Acronyms and abbreviations which have a local scope and therefore are not contained in the [2].

## 2.1 Acronyms and abbreviations

| Abbreviation / Acronym: | Description: |
|---|---|
| SYCL | A C++ based high-level heterogeneous programming model to improve programming productivity on various hardware accelerators. |

**Table 2.1: Acronyms and abbreviations used in the scope of this Document**

## 2.2 Definition of terms

Definition of terms which are not self-explaining and are needed to understand the explanations in this document.

| Terms: | Description: |
|---|---|
| Device | A Device is one of the available hardware accelerators e.g. GPU, FPGA, DSP etc., typically a non-cpu ones, but the CPU also can be considered as Device for computations. |
| Kernel | A function that is executed on a device |
| Host | A Host Device is native computation unit of the ECU. Usually it is a CPU hardware. |
| Single-source programming model | This model allows the kernel code to be embedded in the host code, meaning the host & kernel code in same language in same translation unit. |

**Table 2.2: Definition of terms in the scope of this Document**

# 3 Related Documentation

## 3.1 Input documents & related standards and norms

[1] Design guidelines for using parallel processing technologies on Adaptive Platform
AUTOSAR_AP_EXP_ParallelProcessingGuidelines

[2] Glossary
AUTOSAR_FO_TR_Glossary

[3] Guidelines for using Adaptive Platform interfaces
AUTOSAR_AP_EXP_InterfacesGuidelines

[4] Specification of Manifest
AUTOSAR_AP_TPS_ManifestSpecification

[5] List of Adaptive Platform Functional Clusters
AUTOSAR_AP_TR_FunctionalClusterList

[6] Specification of Platform Health Management
AUTOSAR_AP_SWS_PlatformHealthManagement

[7] Specification of State Management
AUTOSAR_AP_SWS_StateManagement

[8] Specification of Execution Management
AUTOSAR_AP_SWS_ExecutionManagement

[9] Specification of Health Monitoring
AUTOSAR_FO_ASWS_HealthMonitoring

# 4 API Design Visions and Guidelines

## 4.1 The approach

Generally the approach assumes three parts:

- Safe HWA API - specifies set of functions needed for:
  - performing tasks on hardware accelerators
  - performing tasks execution and hardware accelerator state monitoring to provide necessary information to make the appropriate safety measures.
- Vendor implementation - contains actual implementation of Safe HWA API. ISO26262, part 6 should be considered. Generally a vendor implementation can be considered as a wrapper over one or more libraries in order to provide needed functionality.
- Libraries integration - integration of all needed libraries to provide functionality specified in the Safe HWA API. The libraries should implemented according to ISO26262, part 6.

**NOTE**

The major part of the Safe HWA API can be implemented based on SYCL 2020 standard but a vendor can choose any suitable underlying technology.

The general approach is shown on the following figure:

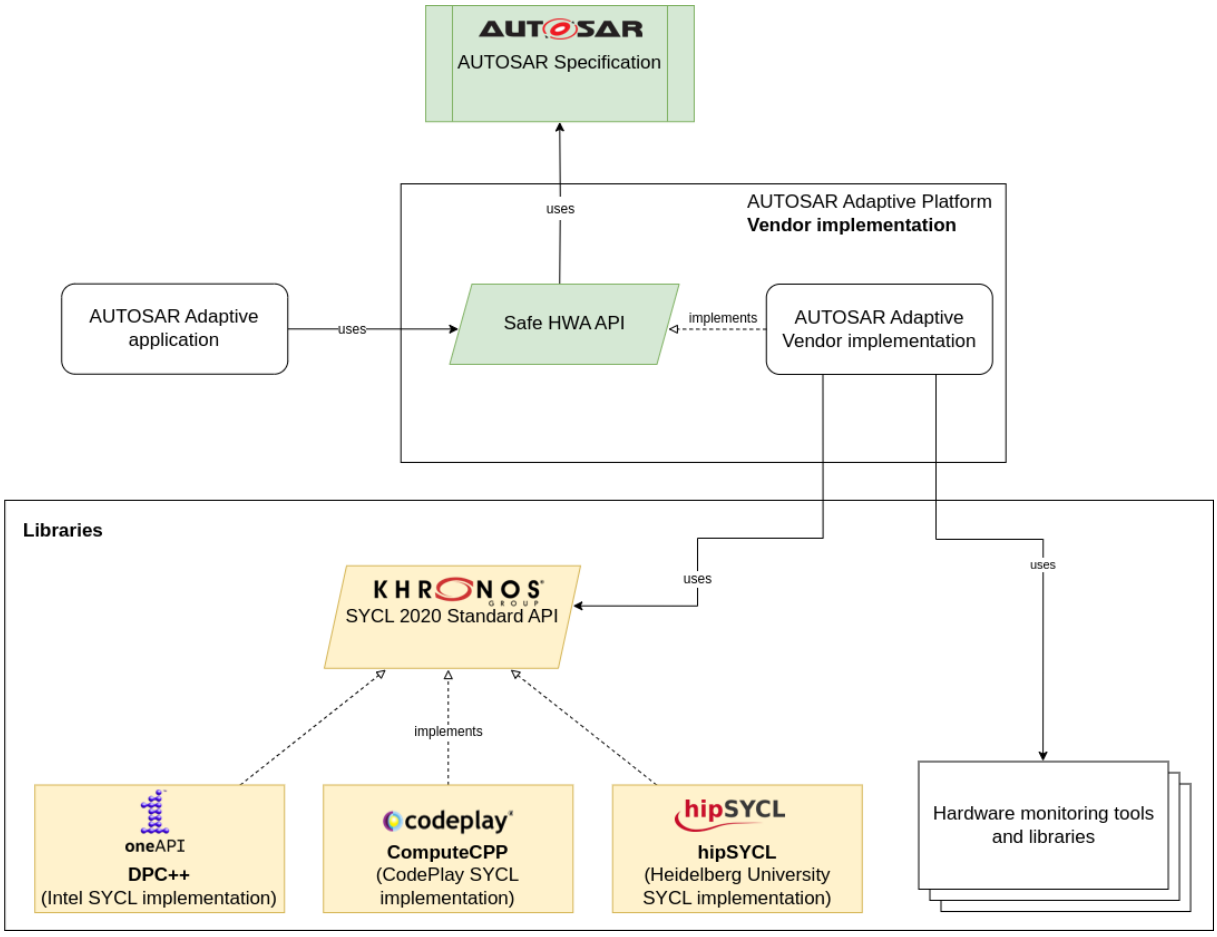


**Figure 4.1: General approach**

## 4.2 The goals of API design

The goal of the API design was to have it as lean as possible. Meaning, that it should only provide the basic set of functionality needed to satisfy the required AUTOSAR Adaptive use-cases like high performance computation for lidar point cloud processing (usually LiDAR can produce 100-300K points for a single Point cloud), sensors fusion, filtering, transforming and other heavy algorithms which can be executed in parallel.

All this could be easily built on top of the open SYCL 2020 Standard API created by Khronos group but it needs to be standardized to support typical collaboration models and portability in AUTOSAR Adaptive.

The second goal for the API design was to have it high-level as much as possible considering re-use of the development approach in AUTOSAR Adaptive platform - so, we base it on C++ language.

The third goal is to standardize the approach considering used rules and requirements for the safety in AUTOSAR Adaptive.

One of the central design points was (as already stated in the introduction) to support ease of the development, having high-level standardized approach considering safety goals. So, you will see in the later chapters, that the application developer, when using Safe HWA API, needs to know modern C++ only, without mandatory knowledges of deep low-level hardware specifics.

The additional functionality provided by the API like hardware monitoring will require utilization of existing monitoring tools provided by the Adaptive platform or integration/utilization of 3rd party libraries, considering safety goals as well.

The API is designed following the next rules:

- Use safe AUTOSAR defined datatypes (e.g. `ara::core::Vector`).
- Will be exception free (named constructors C++ idiom with `ara::core::Result` return values will be used [3]).
- Will have methods for hardware availability monitoring.
- Will have configuration capabilities to limit hardware resources usage according to safety needs of exact system using particular machine manifest.
- Will provide additional control of library methods calls which try to use more hardware resources than allowed by hardware configuration

## 4.3 Functional breakdown

Functionality of Safe HWA API can be split into 4 groups:

- Data storage and management
- Tasks execution
- Devices management and monitoring
- Runtime configuration

These groups are similar for most of the HWA frameworks with small deviations so Safe HWA API will reuse existed experience of HWA frameworks functional distribution. Next chapters will describe these groups and classes used for it.

### 4.3.1 Data storage and management

Since main use-case for `HWA` usage is efficient computation over big sets of data or multiple small sets of data using parallel computing paradigm, Safe HWA API has to provide convenient mechanism for data read/write access on both `HWA` and host `ECU`.

For these purposes Safe HWA API has `Buffer` class which manages data storage and shared access from host `ECU` and `HWA`. But at the same point `Buffer` doesn't have

methods for accessing and modifying stored data, for this goal Safe HWA API contains specific class `Accessor`. This functionality was moved to the separate class for more granular management of access from host `ECU` and `HWA` and preventing redundant data copying.

To make work with data stored inside of the `Buffer` more convenient, Safe HWA API introduces two additional classes - `Range` and `Id`, which will help to choose particular item within the `Buffer` or work with collection of items in `Buffer`. Also, `Range` can be used to define desired size of the `Buffer`.

### 4.3.2 Tasks execution

Next step is to execute needed operations over the stored data. Main class for this part of functionality is `Queue`. Using `Queue` class, it's possible to submit tasks for execution on the `HWA`. After submitting the task to the `Queue` host application (regular AP application run on host `ECU`) will get object of `Event` class which will give possibility to order tasks in correct manner and block some tasks execution before other important tasks will not be finished (will also block appropriate process on host `ECU`).

An important feature of task execution on the `HWA` is error handling. The thing is that tasks submitted to the `HWA` are executed asynchronously (which is good for better performance on host `ECU`), but error handling in this case also must be asynchronous. Therefore, mandatory parameter for `Queue` creation is `AsyncErrorHandler`, callback which will be called when error occurs.

Another important feature is the right choice of appropriate `HWA` for particular task execution. This topic will be described in more detail in the next chapter.

### 4.3.3 Device management and monitoring

For convenient choice of concrete `HWA` for task execution, it's needed to have class representing appropriate `HWA` on the host `ECU` (let's not forget that we don't have an access to the `HWA` driver or some other low-level information). For this purpose, Safe HWA API has `Device` class which incapsulates actual `HWA`. Using of `Device` will give possibility to construct `Queue` for specific `HWA` and check some base characteristics of `HWA`.

Since we can't know which exact `HWA`s are available on the `ECU` on compile time it's important to have possibility to check available `Devices` in runtime and choose needed one. This functionality is incapsulated in `Platform` class, which will give possibility to get all available `Devices`.

Another important part of functionality (especially in scope of safety) is the ability to check the status of `Device` - healthy / not healthy, highly loaded / not loaded etc. This information will give possibility for host application to perform preventive measures and

not run some tasks if determinism of task execution is not guaranteed. Safe HWA API has `DeviceMonitor` class which provides appropriate functionality.

### 4.3.4 Runtime configuration

Important is to prevent not supported resource consumption by AP application. For these purposes capability of machine manifest will be extended with possibility to configure `HWA` resources. Appropriate changes will be integrated to [4].

## 4.4 Interaction with AP application

Safe HWA API will have a number of entities to choose appropriate hardware accelerator, submit and execute tasks on it. There will be available an API for data allocation and processing on the selected device.

AP applications can use `Buffer` to allocate memory and choose which `Device` to use for this allocation. `Queue` allows to submit multiple `TaskHandlers` to be executed on `Device`. Data allocated in `Buffer` can be accessible from `Device` (in order to operate with it on hardware accelerator) if it's bound to the same `Device` as `Queue`.

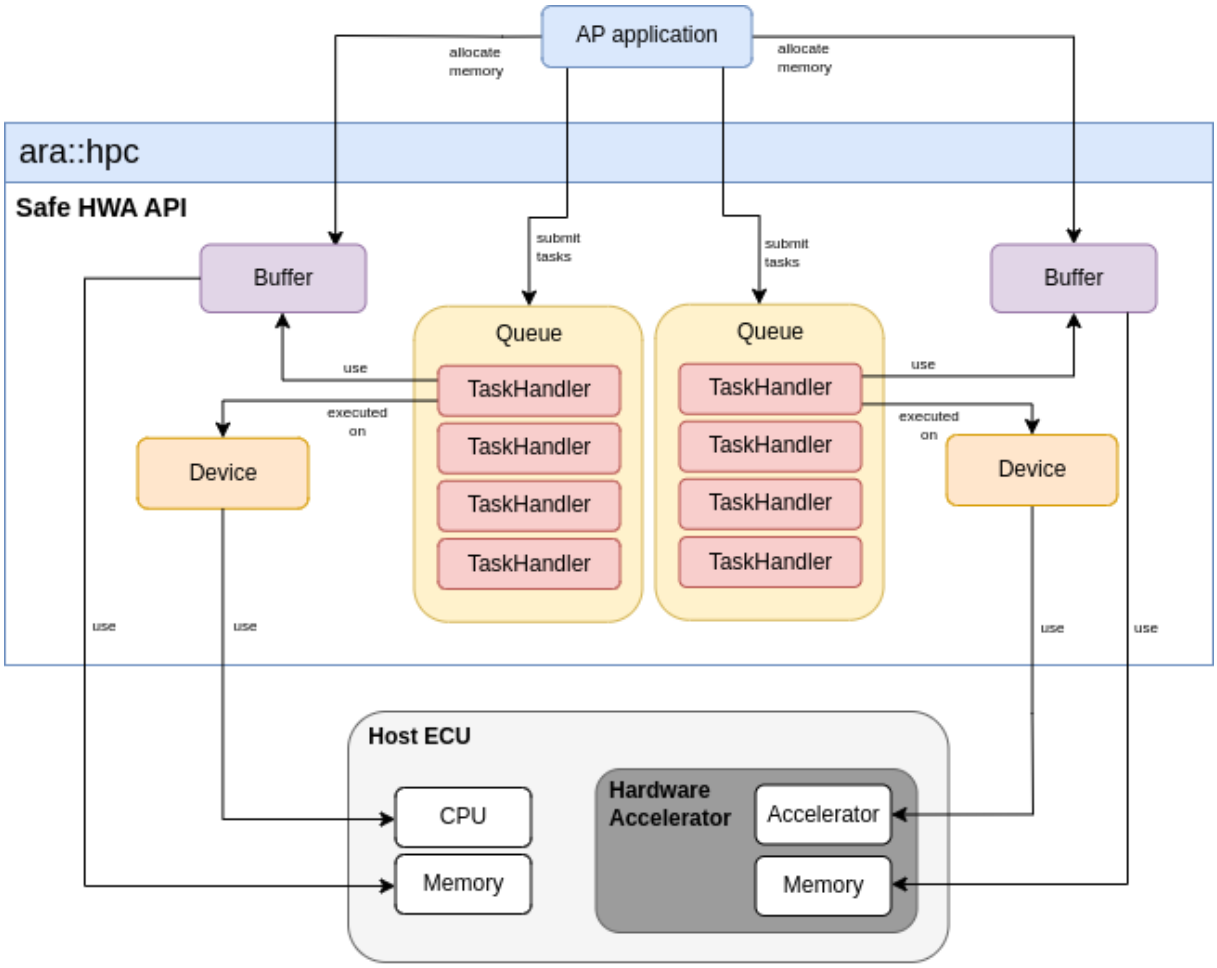


**Figure 4.2: High-level SW Architecture**

It's possible to track the state and explicitly manage the order of `TaskHandlers` execution inside of `Queue` using `Event`, which will be returned after task submit to `Queue`, if `Queue` was constructed as unordered. Otherwise tasks will be executed in order which they were submitted.

# 5 Detailed API description

In this chapter we will go through the Safe HWA API classes, the relations between them and the main methods of each class. It is worth to mention that here we will not show the full interface of each class because it would blow up the document significantly. But we will make precise look at the API elements.

## 5.1 Safe HWA API architecture

First of all lets go through API class diagram to understand the relations between different entities.

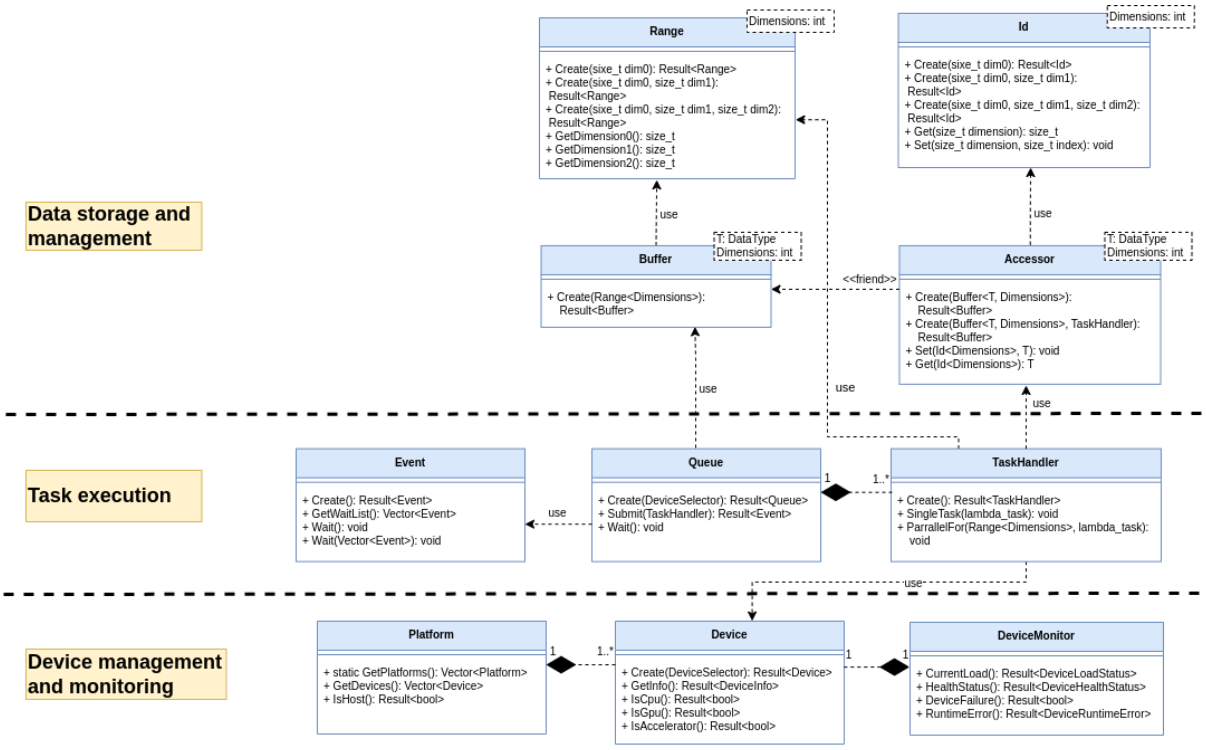


**Figure 5.1: Class diagram**

Important to mention several architectural decisions relevant to all Safe HWA API classes.

- All constructors are private so it's possible to construct objects only using named constructors - static `Create()` methods which are present in each class
- `Create()` methods of classes which are owning data returns `ara::com::SamplePtr` to the object instead of object itself
- Most of return values of methods in Safe HWA API are wrapped in `ara::core::Result` which is described in [3]. So, it's possible check if operation was successful

**NOTE**

Safe HWA API requires new functional cluster and namespace, and it is assumed that it does exist. Temporary shortname `SHWA` is taken for the convenience before official functional cluster introduction in [5]

## 5.2 Queue class

The `Queue` class is designed to collect and handle tasks sent to `Device` for execution. It's possible to construct ordered and unordered `Queue` using `CreateOrdered()` or `CreateUnordered()` factory methods accordingly.

```
using AsyncErrorHandler =
    std::function<void(ara::core::Vector<ara::shwa::Error>)>;
using AsyncErrorList = ara::core::Vector<ara::shwa::Error>;

class Queue final {
    // Create ordered Queue
    static ara::core::Result<ara::com::SamplePtr<Queue>>
        CreateOrdered(const AsyncErrorHandler& errorHandler,
                      const PropertyList& propList = {}) noexcept

    // Create unordered Queue
    static ara::core::Result<ara::com::SamplePtr<Queue>>
        CreateUnordered(const AsyncErrorHandler& errorHandler,
                        const PropertyList& propList = {}) noexcept
}
```

As we can see from previous code snippet, there is one mandatory parameter for all `Queue`s - `AsyncErrorHandler`. Since part of the code will be executed asynchronously in runtime on the `Device`, it's not possible to handle all errors using `ara::core::Result` return value. For such situations we need to set `AsyncErrorHandler`. Basically it's error handling function or functional object which will be called when such error occurs.

**NOTE**

Please keep in mind that `AsyncErrorHandler` can be called not immediately, it depends on scheduling features of particular `Device`.

It's also possible to construct both types of `Queue` for specific `DeviceType` or even for concrete `Device`. Let's view appropriate interfaces for ordered `Queue` only as the interface for unordered `Queue` is absolutely the same.

```
    template <typename DeviceSelector>
    static ara::core::Result<ara::com::SamplePtr<Queue>>
        CreateOrdered(const AsyncErrorHandler& errorHandler,
                      const DeviceSelector& deviceSelector,
                      const PropertyList& propList = {}) noexcept;

    static ara::core::Result<ara::com::SamplePtr<Queue>>
        CreateOrdered(const AsyncErrorHandler& errorHandler,
```

```
                        const Device& device,
                        const PropertyList& propList = {}) noexcept;
```

Key `Queue` functionality is submitting tasks for execution on `Device`. It's being done by using `Submit()` method.

```
    template <typename T>
    ara::core::Result<Event> Submit(T task) noexcept
```

`Submit()` method expects functional objects (functor, lambda function etc.) with `TaskHandler` as input.

```
    queue->Submit([&](ara::shwa::TaskHandler& handler) {
        // Task code
    }
```

**NOTE**

It is important to mention that `Queue` can submit tasks only for accelerator device available on current ECU (physically or virtualized). Any transmitting task to another ECU through `ara::com` is not provided.

For ordered `Queue` it's possible to manage tasks execution order using `Event`s. By default tasks are executed in order they were submitted to the ordered `Queue`. For the unordered `Queue` tasks are executed in most efficient for `Device` order. In this case performance usually better, but AP Application can't control task execution order.

An AP Application is not restricted to one `Queue`, so application developer can create several `Queue`s with different tasks and `Device`s within one Application.

## 5.3 Event class

The `Event` class is designed to provide explicit control of tasks scheduling on `Device` which is available for ordered `Queue`. `Submit()` method of `Queue` class is returning `Event` object and it's possible to call `Wait()` method of `Event` to block AP Application execution until task will be completed.

```
    ara::core::Result<void> Wait() noexcept;
```

There is also static variation of `Wait()` method able to accept wait list(Vector) of `Event`s as an input.

```
    static ara::core::Result<void>
        Wait(const ara::core::Vector<Event>& eventList) noexcept;
```

`Event` Object also provides opportunities to perform task execution monitoring and profiling using `GetInfo()` and `GetProfilingInfo()` methods.

```
    template <typename Info>
    ara::core::Result<Info> GetInfo() const noexcept;

    template <typename ProfilingInfo>
    ara::core::Result<ProfilingInfo> GetProfilingInfo() const noexcept;
```

## 5.4 Buffer class

`Buffer` class is designed to own data available on both - host (AP Application) and device (hardware accelerator, e.g. GPU). `Buffer` is a template class and have to be created with defined data type and buffer dimensions (max is 3-dimensional buffer).

```
template <typename T, int Dimensions = 1,
typename AllocatorT = BufferAllocator<std::remove_const_t<T>>>
class Buffer final
{
    static ara::core::Result<ara::com::SamplePtr<Buffer>>
        Create(const Range<Dimensions>& bufferRange,
               const PropertyList& propList = {}) noexcept

    static ara::core::Result<ara::com::SamplePtr<Buffer>>
        Create(const Range<Dimensions>& bufferRange,
               AllocatorT allocator,
               const PropertyList& propList = {}) noexcept;

    // other methods

};
```

`Buffer` doesn't have own methods to access hosted data. For accessing data inside the `Buffer`, `Accessor` class is used.

## 5.5 Accessor class

As was stated before, the `Accessor` class is designed to access data hosted by `Buffer`. `Accessor` has very tight relation to `Buffer` and provide three different capabilities:

- access to the data managed by a buffer
- access to local memory on a device
- define the requirements to memory objects which determine the scheduling of kernels

The `Accessor` provides access to the `Buffer` data in 3 modes - read-only, write-only and read-write.

```
enum class AccessMode
{
    read, // Read-only access.
    write, // Write-only access.
    read_write // Read and write access.
};
```

The `Accessor` can be created only with defined data type and dimensions like `Buffer`. In addition to it for `Accessor` need to be specified template parameters for access mode and target device.

```
template <typename T,
    int Dimensions = 1,
    AccessMode AccessModeType = AccessMode::read,
    Target AccessTargetType = Target::device,
    typename AllocatorT = BufferAllocator<std::remove_const_t<DataT>>>
class Accessor final
{
    // Accessor methods
};
```

Since `Accessor` can act only in conjunction with `Buffer` so it has to be constructed with specified `Buffer` object. Also it's possible to specify `TaskHandler` if `Accessor` is created within the task.

```
static ara::core::Result<ara::com::SamplePtr<Accessor>>
    Create(const ara::com::SamplePtr<Buffer<T, Dimensions>>& bufferRef)
        noexcept;

static ara::core::Result<ara::com::SamplePtr<Accessor>>
    Create(const ara::com::SamplePtr<Buffer<T, Dimensions>>& bufferRef,
           TaskHandler& handler) noexcept;
```

Accessor objects must always be constructed in host code, either in command group scope or in application scope. Whether the constructor blocks waiting for data to synchronize depends on the type of accessor. Those accessors which provide access to data within a command do not block. Instead, these accessors define a requirement which influences the scheduling of the command. Those accessors which provide access to data from host code do block until the data is available on the host.

It is worth to note here that accessors can be created with different access modes and most accessors have an `AccessMode` template parameter, which specifies whether the accessor can read or write the underlying data. This information is used by the runtime when defining the requirements for the associated command, and it tells the runtime whether data needs to be transferred to or from a device before data can be accessed through the accessor. Accessors provide the runtime with information about how we plan to use the data in buffers, allowing it to correctly schedule data movement.

To access `Buffer` data `Accessor` overloads `operator[]` with `Id` as input. It gives access to particular element from `Buffer`. Also application developer can use alternative `Set()`/`Get()` methods.

```
Reference<DataT> operator[](Id<Dimensions> index) const;

void Set(Id<Dimensions> index, DataT value);

DataT Get(Id<Dimensions> index);
```

## 5.6 Range class

The `Range` class is designed to define up to 3-dimensional range which is required for queues, buffers and accessors. The `Range` is a lightweight class needed to support

interfaces of other classes. For `Buffer` and `Queue` it can be used to define size of the `Buffer` or range of access to `Buffer` data. For `Queue` and `TaskHandler` it can be used to perform iteration within `Buffer` data.

`Range` have to be constructed with defined `Dimensions`. For this purpose it has `Dimensions` template parameter and 3 variants of named constructor.

```
template <int Dimensions = 1>
class Range final
{
    static ara::core::Result<Range>
        Create(std::size_t dim0) noexcept;

    static ara::core::Result<Range>
        Create(std::size_t dim0,
               std::size_t dim1) noexcept;

    static ara::core::Result<Range>
        Create(std::size_t dim0,
               std::size_t dim1,
               std::size_t dim2) noexcept;
};
```

## 5.7 Id class

The `Id` class is designed to define particular item index within `Range`. It has very similar interface to `Range` interface, but semantically defines particular position inside the `Range`.

```
template <int Dimensions = 1>
class Id final
{
    static ara::core::Result<Id>
        Create(std::size_t dim0) noexcept;

    static ara::core::Result<Id>
        Create(std::size_t dim0,
               std::size_t dim1) noexcept;

    static ara::core::Result<Id>
        Create(std::size_t dim0,
               std::size_t dim1,
               std::size_t dim2) noexcept;
};
```

`Id` is actively used for iterations inside of the `Buffer`.

## 5.8 Device class

The `Device` class is designed to represent a hardware accelerator device. It's needed to define particular computation unit choosen to execute tasks from `Queue`.

`Device` can be created with `DeviceSelector` argument, which will create `Device` entity for particular type of computation unit.

```
template <typename DeviceSelector>
static ara::core::Result<ara::com::SamplePtr<Device>>
    Create(const DeviceSelector& deviceSelector) noexcept;
```

`DeviceSelector` marker which helps to specify correct device type. `DeviceSelector` is designed as stand-alone class hierarchy to improve type safety.

```
class DeviceSelector;

class DefaultSelector:      public DeviceSelector;
class GpuSelector:          public DeviceSelector;
class CpuSelector:          public DeviceSelector;
class AcceleratorSelector:  public DeviceSelector;
class HostSelector:         public DeviceSelector;
```

In case of `DefaultSelector` usage, default `Device` will be used. Default `Device` can be set in run parameters.

Also `Device` has methods to check nature of actual `Device` object.

```
ara::core::Result<bool> IsCpu() const noexcept;
ara::core::Result<bool> IsGpu() const noexcept;
ara::core::Result<bool> IsAccelerator() const noexcept;
```

## 5.9 Device monitor class

The `DeviceMonitor` class is designed for monitoring of current state of particular `Device`. It can help to understand if device is ready to perform more tasks and make correct decision for AP application. For this purposes `DeviceMonitor` has `CurrentLoad()` and `HealthStatus()` methods.

```
ara::core::Result<DeviceLoadStatus> CurrentLoad()
    const noexcept;
ara::core::Result<DeviceHealthStatus> HealthStatus()
    const noexcept;
```

It's also possible to trace device failures using `DeviceFailure()` and `RuntimeError()` methods.

```
ara::core::Result<bool> DeviceFailure()
    const noexcept;
ara::core::Result<DeviceRuntimeError> RuntimeError()
    const noexcept;
```

## 5.10 Task handler class

The `TaskHandler` class is designed as class contains number of methods which can be performed on `Device`.

```
template <typename T>
void SingleTask(T cgf) noexcept;

template <typename T>
void ParallelFor(std::size_t bufferSize, T cgf) noexcept;
```

`TaskHandler` can't be used outside of `Queue::Submit()` functionality, it's useful only inside submitted task.

```
queue->Submit([&](ara::shwa::TaskHandler& handler) {

    // Accessing Buffer

    handler.ParallelFor(bufferSize, [&](ara::shwa::Id<1> id) {
        int random_value = 0;
        accessor.Value()->Set(id, random);
    });
}
```

## 5.11 Platform class

The `Platform` class is designed to represent platform which contains a set of devices or host device. It's possible to get available on platform devices or check for host device next methods can be used.

```
ara::core::Vector<ara::com::SamplePtr<Device>> GetDevices()
    noexcept;

ara::core::Result<bool> IsHost()
    noexcept;
```

# 6 Explanation and examples for Application developers

## 6.1 Base scenario example

In this section we want to go through base scenario of Safe HWA API usage which will show main API components.

```
#include "ara/com/sample_ptr.h"

#include "ara/shwa/accessor.h"
#include "ara/shwa/buffer.h"
#include "ara/shwa/id.h"
#include "ara/shwa/range.h"
#include "ara/shwa/task_handler.h"
#include "ara/shwa/queue.h"

int main(int argc, char* argv[])
{
    using BufferDataT = int;

    // Creating async error handler
    auto error_handler =
        [&](ara::shwa::AsyncErrorList error_list) {
            // Async error handling logic
        });

    // Creating queue
    ara::core::Result<ara::com::SamplePtr<ara::shwa::Queue>>
        queue = ara::shwa::Queue::CreateOrdered(error_handler);

    // Creating one-dimentional range for 4 elements
    constexpr int bufferDimension(1);
    constexpr int bufferSize(4);
    ara::core::Result<ara::shwa::Range<bufferDimension>> range
        = ara::shwa::Range<bufferDimension>::Create(bufferSize);

    // Creating buffer of 4 ints
    ara::core::Result<ara::com::SamplePtr<
        ara::shwa::Buffer<int, bufferDimension>>> buffer
            = ara::shwa::Buffer<BufferDataT, bufferDimension>
                ::Create(range.Value());

    // Submit some work to be done in Queue
    queue.Value()->Submit([&](ara::shwa::TaskHandler& handler) {
        ara::core::Result<
            ara::com::SamplePtr<ara::shwa::Accessor<
                BufferDataT, bufferDimension, ara::shwa::AccessMode::read
    >>>
                    accessor = ara::shwa::Accessor<
                        BufferDataT, bufferDimension,
                        ara::shwa::AccessMode::read>
                            ::Create(buffer.Value(), handler);

```

```
        handler.ParallelFor(bufferSize, [&](ara::shwa::Id<1> id) {
            int index = id.Get(0);
            accessor.Value()->Set(id, 10 * index);
        });

        handler.SingleTask([&]() {
            ara::core::Result<ara::shwa::Id<1>> idRes
                = ara::shwa::Id<1>::Create(0);

            ara::shwa::Id<1> id = idRes.Value();

            for (std::size_t i = 0; i < bufferSize; ++i) {
                id.Set(0, i);
                int value = accessor.Value()->Get(id);
            }
        });
    });
}
```

### 6.1.1 Code executed on the Host

Let's go through this code step by step. Skipping trivial parts like includes, in first section we can see `Queue` creation. In this case we see default `Queue` creation, without binding to specific `Device` (default `Device` will be used). But we still have to set error handler to catch asynchronous errors.

```
    // Creating async error handler
    auto error_handler =
        [&](ara::shwa::AsyncErrorList error_list) {
            // Async error handling logic
        });

    // Creating queue
    ara::core::Result<ara::com::SamplePtr<ara::shwa::Queue>>
        queue = ara::shwa::Queue::CreateOrdered(error_handler);
```

Next thing we do is creating `Range` to define size of our future `Buffer`. We will create one-dimensional `Range` and define size for 4 elements. `Buffer` with one-dimensional `Range` acts as regular C++ array. It's also possible to create 2 or 3 dimensional `Range`.

```
    // Creating one-dimensional range for 4 elements
    constexpr int bufferDimension(1);
    constexpr int bufferSize(4);
    ara::core::Result<ara::shwa::Range<bufferDimension>> range
        = ara::shwa::Range<bufferDimension>::Create(bufferSize);
```

Now let's discuss `Buffer` creation. First we define type of data which Buffer will operate with. And then creating `Buffer` itself.

```
    using BufferDataT = int;

    // ...

```

```
    ara::core::Result<ara::com::SamplePtr<
        ara::shwa::Buffer<int, bufferDimension>>> buffer
            = ara::shwa::Buffer<BufferDataT, bufferDimension>
                ::Create(range.Value());
```

After that we are ready to submit our task to the `Queue` and the `Queue` will manage scheduling this task to be performed on `Device`. In the current example we use non-blocking task submission. In order to block the host code execution we need to use `Wait()` operation. This is last code portion which will be executed on the host.

```
    // Submit some work to be done in Queue
    queue.Value()->Submit([&](ara::shwa::TaskHandler& handler) {
        // ...
    });
```

### 6.1.2 Code executed on the Device

Now let's explore the code which will be executed on `Device`

As a first step we are creating `Accessor`. `Accessor` will give us possibility access `Buffer` data on the `Device`. In this example we are creating `read` accessor, which is defined by template parameter `ara::shwa::AccessMode::read`. Also important to mention that we bind our access to the `handler`, so this accessor can only be used within this particular handler.

```
ara::core::Result<
    ara::com::SamplePtr<ara::shwa::Accessor<
        BufferDataT, bufferDimension, ara::shwa::AccessMode::read>>>
            accessor = ara::shwa::Accessor<
                BufferDataT, bufferDimension, ara::shwa::AccessMode::read>
                    ::Create(buffer.Value(), handler);
```

Having `Accessor` we can execute some operations with `Buffer` data on `Device`. First shown type of operation is `ParallelFor()`. In this example we are simply iterating over the `Buffer` and multiplying each value by `10`.

```
handler.ParallelFor(bufferSize, [&](ara::shwa::Id<1> id) {
    int index = id.Get(0);
    accessor.Value()->Set(id, 10 * index);
});
```

Next type of operation is `SingleTask()`. In `SingleTask()` we can do some operation with single data entity or also iterate over whole `Buffer`, but here we need to manage `Id` manually.

```
handler.SingleTask([&]() {
    ara::core::Result<ara::shwa::Id<1>> idRes
        = ara::shwa::Id<1>::Create(0);

    ara::shwa::Id<1> id = idRes.Value();

    for (std::size_t i = 0; i < bufferSize; ++i) {
```

```
        id.Set(0, i);
        int value = accessor.Value()->Get(id);
    }
});
```

## 6.2 Accessors management

As `Buffer`s are not directly accessed by the program and are instead used through accessor objects, it is worth to recall that the data actually is stored in `Buffer` but to access the stored data it is required to utilize `Accessor` object.

The `Accessor` object creation can look like on the code snippet below:

```
ara::core::Result<
    ara::com::SamplePtr<ara::shwa::Accessor<
        BufferDataT, bufferDimension, ara::shwa::AccessMode::read>>>
            accessor = ara::shwa::Accessor<
                BufferDataT, bufferDimension, ara::shwa::AccessMode::read>
                    ::Create(buffer.Value(), handler);
```

Accessors allows to get and set values in the buffer by index, having appropriate functions. Set value by index - example:

```
handler.ParallelFor(bufferSize, [&](ara::shwa::Id<1> id) {
    int index = id.Get(0);
    accessor.Value()->Set(id, 10 * index); // Accessor Set(..) function
    call
});
```

Get value by index - example:

```
handler.SingleTask([&]() {
    ara::core::Result<ara::shwa::Id<1>> idRes
        = ara::shwa::Id<1>::Create(0);

    ara::shwa::Id<1> id = idRes.Value();

    for (std::size_t i = 0; i < bufferSize; ++i) {
        id.Set(0, i);
        int value = accessor.Value()->Get(id); // Accessor Get(..) function
     call
    }
});
```

Below you can observe simple but complete example of accessor usage.

```
#include "ara/core/result.h"
#include "ara/com/sample_ptr.h"

#include "ara/shwa/accessor.h"
#include "ara/shwa/buffer.h"
#include "ara/shwa/id.h"
#include "ara/shwa/range.h"
#include "ara/shwa/task_handler.h"
```

```
#include "ara/shwa/queue.h"
#include <array>

using namespace ara::shwa;

constexpr int N = 42;

int main() {

std::array < int, N > a, b, c;
for(int i = 0; i < N; i++) {
    a[i] = b[i] = c[i] = 0;
}

// Creating async error handler
auto error_handler = [&](ara::shwa::AsyncErrorList error_list) {
    // Async error handling logic
});

// Creating queue
ara::core::Result < ara::com::SamplePtr < ara::shwa::Queue >> queue =
    ara::shwa::Queue::CreateOrdered(error_handler);

constexpr int bufferDimension(1);

ara::core::Result<shwa::Range<bufferDimension>> range =
    shwa::Range<bufferDimension>::Create(N);

// Buffers creation. Separate for each array
ara::core::Result <ara::com::SamplePtr<shwa::Buffer<int, bufferDimension>>>
    buffer_a = shwa::Buffer<BufferDataT, bufferDimension>
        ::Create(range.Value(), a);

ara::core::Result<ara::com::SamplePtr<shwa::Buffer<int, bufferDimension>>>
    buffer_b = shwa::Buffer<BufferDataT, bufferDimension>
        ::Create(range.Value(), b);

ara::core::Result<ara::com::SamplePtr<shwa::Buffer<int, bufferDimension>>>
    buffer_c = shwa::Buffer<BufferDataT, bufferDimension>
        ::Create(range.Value(), c);

// Submitting first task
queue.submit([&](shwa::TaskHandler& handler) {
    ara::core::Result<ara::com::SamplePtr<ara::shwa::Accessor<BufferDataT,
        bufferDimension, shwa::AccessMode::read>>> // Read Only accessor
            accessor_a = shwa::Accessor<BufferDataT, bufferDimension,
                shwa::AccessMode::read>::Create(buffer_a.Value(), handler);

    ara::core::Result<ara::com::SamplePtr<shwa::Accessor<BufferDataT,
        bufferDimension, shwa::AccessMode::write>>> // Write Only accessor
            accessor_b = shwa::Accessor<BufferDataT, bufferDimension,
                shwa::AccessMode::write>::Create(buffer_b.Value(), handler)
    ;

    // compute B
    handler.parallel_for(N, [=](id <1> i) {
```

```
64          *(accessor_b)[i] = *(accessor_a)[i] + 1;
65      });
66  });
67
68  // Submitting second task
69  queue.submit([&](shwa::TaskHandler& handler) {
70      ara::core::Result<ara::com::SamplePtr<shwa::Accessor< BufferDataT,
71          bufferDimension, shwa::AccessMode::read>>> // Read Only accessor
72              accessor_a = shwa::Accessor<BufferDataT, bufferDimension,
73                  shwa::AccessMode::read>::Create(buffer_a.Value(), handler);
74
75      // read A
76      handler.parallel_for(N, [=](id <1> i) {
77          // Useful only as an example
78          int data = *(accessor_a)[i];
79      });
80  });
81
82  // Submitting third task
83  queue.submit([&](shwa::TaskHandler& handler) {
84      // RAW of buffer_b
85      ara::core::Result<ara::com::SamplePtr<ara::shwa::Accessor<BufferDataT,
86          bufferDimension, shwa::AccessMode::read>>> // Read Only accessor
87              accessor_b = shwa::Accessor<BufferDataT, bufferDimension,
88                  shwa::AccessMode::read>::Create(buffer_b.Value(), handler);
89
90      ara::core::Result<ara::com::SamplePtr<shwa::Accessor<BufferDataT,
91          bufferDimension, shwa::AccessMode::write>>> // Write Only accessor
92              accessor_c = shwa::Accessor<BufferDataT, bufferDimension,
93                  shwa::AccessMode::write>::Create(buffer_c.Value(), handler)
      ;
94
95      // compute C
96      handler.parallel_for(N, [=](id < 1 > i) {
97          *(accessor_c)[i] = *(accessor_b)[i] + 2;
98      });
99  });
100
101 return 0;
102
103 }
```

## 6.3 Device management approaches

Using Safe HWA API we can work with `Device`s in several ways. The most straightforward way is to use `DeviceSelector`.

```
1  ara::core::Result<ara::com::SamplePtr<ara::shwa::Device>> device =
2      ara::shwa::Device::Create(ara::shwa::GpuSelector);
```

Created `Device` can be later used for `Queue` creation. Alternatively `Queue` can be created with `DeviceSelector` or without specifying any `Device` at all, in this case default device will be used.

```
auto error_handler =
    [&](ara::shwa::AsyncErrorList error_list) {
        // Async error handling logic
    });

ara::core::Result<ara::com::SamplePtr<ara::shwa::Queue>>
    queue_for_cpu
        = ara::shwa::Queue::CreateOrdered(error_handler, ara::shwa::
    CpuSelector);

ara::core::Result<ara::com::SamplePtr<ara::shwa::Queue>>
    queue_for_default
        = ara::shwa::Queue::CreateOrdered(error_handler);
```

For such approaches we can check which exact device is used by `Queue`

```
std::cout
    << "Device used: "
    << queue.Value()->GetDevice().GetInfo<ara::shwa::info::device::name>()
    << std::endl;
```

Also `Device` has additional methods to define `Device` nature - `IsCpu()`, `IsGpu()` and `IsAccelerator()`

```
if (device.IsGpu() == true)
{
    std::cout << "Acting on GPU Device" << std::endl;
}
```

For some cases we need to examine existed capabilities on current platform and make the most efficient decision. For this case we can get all available `Device`s from `Platform`.

```
ara::core::Result<ara::shwa::Device> FindGPUDevice(
    ara::shwa::Platform platform)
{
    ara::core::Result<ara::shwa::Device> gpu_device_result;
    for (auto device : platform.GetDevices())
    {
        if (device.IsGpu() == true)
        {
            gpu_device_result
                = ara::core::Result<ara::shwa::Device>::FromValue(device);

            break;
        }
    }

    return gpu_device_result;
}
```

## 6.4 Error handling

There are two mechanisms to handle errors using Safe HWA API - synchronous and asynchronous.

**NOTE**

There is one more important error handling mechanism - using PHM and reporting checkmarks. It's not a part of Safe HWA API, but general error handling mechanism for all AUTOSAR AP. This topic will be described in more details in chapter 7 `Safety Approach Explanation for Application developers`

Synchronous errors are handled by processing operation return value. Most of return values in Safe HWA API are wrapped into `ara::core::Result`. Using it application developer can check result of operation - if it contains an `Error` or `Value`. It's possible to get a `Value` if there is no `Error` or check exact `Error` in opposite case.

```
using BufferDataT = int;
constexpr int bufferDimension(1);

ara::core::Result<ara::shwa::Range<bufferDimension>> range_res
    = ara::shwa::Range<bufferDimension>::Create(16);

ara::core::Result<ara::com::SamplePtr<
    ara::shwa::Buffer<int, bufferDimension>>>
        buffer_res = ara::shwa::Buffer<BufferDataT, bufferDimension>
            ::Create(range_res.Value());

if (buffer_res.HasValue()) {
    auto buffer = buffer_res.Value();
    // proceed with Buffer handling
} else {
    auto buffer_err = buffer_res.Error();
    // provide error handling
}
```

Asynchronous error handling method is used to handle errors occurred inside of submitted task, since it's not possible to handle it synchronously. To catch this kind of error we need to use `AsyncErrorHandler` which we set during `Queue` creation. Handler function bound to `AsyncErrorHandler` will be called (obviously asynchronously) if any errors will occur inside of submitted task.

```
// Create error handler
auto error_handler =
    [&](ara::shwa::AsyncErrorList error_list) {
        // Async error handling logic
    });

// Create Queue
ara::core::Result<ara::com::SamplePtr<ara::shwa::Queue>>
    queue = ara::shwa::Queue::CreateOrdered(error_handler);

// Submit task to the Queue,
// all inner errors will be caught by error_handler
```

```
queue.Value()->Submit(
    [&](ara::shwa::TaskHandler& handler) {
        // Task logic
    });
```

## 6.5 Device state monitoring

For the reasons of correct and robustness AP application performing, it's important to know state of `Device` before submitting task there. For this purposes Safe HWA API has `DeviceMonitor` class which can provide general information about `Device` like `Load` and `HealthStatus`.

```
// Creating Queue and Device
// ...

// Creating DeviceMonitor
ara::core::Result<ara::shwa::DeviceMonitor> monitor
    = ara::shwa::DeviceMonitor::Create(device);

// Checking Device state
auto device_health = monitor.Value().HealthStatus();
auto device_load = monitor.Value().CurrentLoad();

if (device_health.HasValue() && device_load.HasValue()
    && device_health.Value() != ara::shwa::DeviceHealthStatus::None
    && device_health.Value() != ara::shwa::DeviceHealthStatus::Faulty
    && device_load.Value() <= ara::shwa::DeviceLoadStatus::Percents70)
{
    // Performing planned tasks
    queue.Value()->Submit(
        [&](ara::shwa::TaskHandler& handler) {
            // Task logic
        });
}
else
{
    // Performing application corrective measures
}
```

In code example above we can see that we are creating `DeviceMonitor` to check if `Device` is in appropriate state for performing needed tasks. And only after that we are submitting task itself for execution. Also it's possible to provide corrective measures if `Device` is not ready for performing current task - choose another task for execution, skip an iteration and try once more, report application health status etc.

# 7 Safety Approach Explanation for Application developers

## 7.1 API Safety explanation

The API on its own can not provide the required safety level. It can be achieved in conjunction with the other entities in the entire safety stack, like runtime libraries, compiler, hardware driver and the hardware itself. The API it is just one part of the Safety stack.

In order to get higher level of the safety, it is recommended for application developers to utilize existing AUTOSAR Adaptive Safety related Mechanisms like Platform Health Management(PHM) [6], State Management(SM) [7] and Execution Management(EM) [8]. Also, Hardware monitoring is required to gather all needed information to proceed safely. Hardware monitoring functions are provided by the API.

## 7.2 Safe HWA API utilization rules

The following rules should be considered during application development:

- The application should use Hardware Monitoring (provided by the API) to handle potential HW errors internally and gather necessary hadware status information to provide the health status via checkpoints
- The application developer should define upper bounds(maximum time) for waiting for the device response and use software timers for checks that the kernel function is executed on the device within defined time range. Otherwise this situation should be considered as device failure and appropriate measures should be performed. The upper bounds can be defined based on application criticality, meaning how long can the application wait for the device response.
- The application should be registered as Supervised Entity (SE) within Platform Health Management system. [6]
- The application should report its checkpoints (a point in control flow of a Supervised Entity, where the activity is reported) to Platform Health Management. See [6]
- The application should be part of appropriate Function Group (to allow application start or termination by Execution Management)
- The application should be monitored by PHM via appropriate Health Channel Supervision. [6]
- The application should be running under Health Monitoring performed by PHM. [6]
- Application developer should configure appropriate checkpoints for Alive, Deadline and Logical Supervisions. See [9]

- The Elementary Supervision Status(current state of an Alive Supervision, Deadline Supervision or Logical Supervision) based on the evaluation (correct/incorrect) of the supervision, should be performed. [6]

  **NOTE**

  The Logical Supervision is required by the safety standards(IEC61508 or ISO26262)

## 7.3 Additional Safety concerns - Error handling

It is worth to mention that 2 types of error handling should be applied:

- Internal error handling: In this case application developer handles the errors himself within the application and decides what actions should be performed.
- External error handling: In this case the application developer should use existing safety related mechanisms like checkpoints reporting to the PHM, In this case the Health Monitoring initiates mechanisms to recover from supervision failures. These range from notifying a central error handler to a global reset of the ECU. For the details see [9]

# A Appendix

No content