Document Title	Demonstrator Design of Functional Cluster Time Synchronization
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	905

Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R20-11

Document Change History

Date	Release	Changed by	Description
2020-11-30	R20-11	AUTOSAR Release Management	• Known Limitations chapter clean-up
2019-11-28	R19-11	AUTOSAR Release Management	• Removed generated API tables • Changed Document Status from Final to published
2019-03-29	19-03	AUTOSAR Release Management	• Adapt to major changes in the SWS_TimeSynchronization
2018-11-02	18-10	AUTOSAR Release Management	• Architecture clean-up • Utilizing core types • findResource added
2018-05-02	18-03	AUTOSAR Release Management	• Initial release

Disclaimer

Table of Contents

1 Introduction

This document describes the design (approach and decisions) of the Functional Cluster Time Synchronization (tsync) for AUTOSAR's Adaptive Platform Demonstrator.

The decisions taken for the AUTOSAR Adaptive Platform Demonstrator may not apply to other implementations as the standard is defined by the specifications. Nevertheless, the demonstrator may supplement and ease the understanding of the specifications.

The main goal for development of the Time Synchronization demonstrator code was to provide a proof of concept for time synchronization features specified in AUTOSAR Adaptive.

1.1 Known limitations

This reference implementation does not yet include all the features mentioned in the current standard, or there are known issues that still need to be fixed in the standard but are already incorporated in the code:

- It is still to be discussed, if TBs shall be re-configurable. Demonstrator allows reconfiguration.
- Any implementation of setTime() in SynchMasterTB and OffsetMasterTB is lacking the bus transmission feature.
- getTimeLeap() in TimeBaseStatus is lacking the implementation.
- getUserData() in TimeBaseStatus currently returns only an empty byte-vector.
- SWS_TS_00102 is currently not checked in implementation of setTime()/update Time() in SyncMasterTB class.
- setLatestClockValue() and setLatestClockUpdate() in clock_data.h should be merged in order to ensure, that both values are set at the same time.
- The TSync example applications in sample-applications/tsync-example/ work only locally (since the implementation for transmission of time information over bus is missing).

1.2 Deviations from specification

The current implementation of time synchronization in the AUTOSAR Adaptive Platform Demonstrator is aligned with the SWS released in 19-03.

1.3 Used OSS components

This sections lists all OSS components used in the implementation of the Functional Cluster, their architectural context and the rationale for their use.

OSS	Version	Explanation	Binding	License	License Version
Google Test	1.8.0	Unit testing	Not relevant	3-Clause BSD	NA

2 Overview on architecture

2.1 Design approach / Design principles

The design goal of the Time Synchronization API for AUTOSAR adaptive applications is to provide type-safe abstraction of time values in a std::chrono like manner. The end users shall be able to use arithmetic functions and converters provided by std::chrono with their own clocks. Since std::chrono is almost completely based on static members and compile time evaluation, the time sync API shall provide similar behaviour. This also aids the reduction of latency by shifting operations from runtime to compile time.

2.2 Overview on architecture

The public API consists of the following modules:

- ClockData classes (abstraction layers for time information stored in the system in a shared location. POD –> strong types)
- Common (helper functions to enable bitwise operations on strongly typed enums)
- TB classes (the clock type specific abstraction of time information)
- TimeBaseStatus class (container that is able to create and store snapshots of a clocks time info)
- TimeBaseConfig class (a universal container for time base configurations)

2.3 Class Overview

2.3.1 RawClockData

A class that is used to read and write clock data from or to shared memory or any other shared location.

2.3.2 ClockData

ClockData class is used to provide specific abstraction of clock data stored somewhere in the system. PODs are translated into clock specific time_points and strongly typed enums.

2.3.3 PureLocalTB

A PureLocalTB is a locally managed time base. The time information is only available for those applications that have direct access to the same memory, since time information is not distributed over busses.

2.3.4 SynchMasterTB

A SynchMasterTB is used to publish local time information to other entities in the system. A SynchMasterTB is using a reference clock to maintain its time. The "tick count" per reference clock quantum can be adjusted by setting the rate correction to value between] 0.0 , INF [.

2.3.5 SynchSlaveTB

A SynchSlaceTB is the local representation of a foreign clock.

2.3.6 OffsetMasterTB

An OffsetMasterTB is supposed to be used on the clock-master side to provide different time values based on the same time source. One possible use-case would be a UTC provider that has one single hardware source for the actual clock value, but still provides the correct time for every time zone over the busses. This results in increased bus load, but lowers the requirements towards free running timers on system level.

2.3.7 OffsetSlaveTB

An OffsetSlaveTB is able to achieve the same functionality like the OffsetMasterTB, but without the additional overhead on the busses. An OffsetSlaveTB is based on a Synch SlaveTB and its only job is to add an offset to the reference clock value. For example UTC is configured as a SynchSlaveTB and for each time zone there is a different Offset SlaveTB.

2.3.8 TimeBaseConfig

A TimeBaseConfig object is supposed to be the one universal object that is able to configure any type of time base. It contains all the information that is necessary to configure a time base object, but does not contain any TB related types.

2.3.9 TimeBaseStatus

The TimeBaseStatus class is currently the most complex one that is provided by the time synchronization implementation. Reason behind is the requirement of a common TimeBaseStatus that provides a different implementation when being used with different time bases. An OffsetTB is supposed to create TimeBaseStatuses that return a TimeBaseStatus of the underlying SynchTB on call of the getSynchStatus() method. On the other hand calling this method on a TimeBaseStatus created by a SynchTB will return a copy of the status.

To achieve the functionality described above, a combination of polymorphism and template specialization is used. The TimeBaseStatus class provides the common feature set, that is independent of the getSynchStatus() implementation. The specific features are inherited from the TimeBaseStatusImpl. There are two versions of this class defined in the time_base_status header. The compiler will automatically choose the right one based on the template parameters of the TimeBaseStatus. For SynchTBs both the TB and STB parameter are identical, which will lead to inheritance of features specified in the TimeBaseStatusImpl that only takes one template parameter. This implementation does not fit for OffsetTBs and the compiler will use the version that takes two template parameters in this case.

Beside the difference in how the getSynchStatus() is implemented, both versions of the TimeBaseStatusImpl need to provide their own version of the comparison operator.

The update counter is also moved to the TimeBaseStatusImpl, because its value depends on the TB-type. For OffsetTBs the counter value shall represent the one of the underlying SynchTB, whilst for SynchTBs the value is maintained internally.

2.3.10 TimeBaseType

The TimeBaseType enum is used to distinguish between different types of time bases.

2.3.11 Identities

The enumerations SynchMasterIdentity, PureLocalIdentity, SynchSlaveIdentity, OffsetMasterIdentity and OffsetSlaveIdentity are used to create different types for each manifestation of a time base. This does not affect the global scope of time synchronization and is only used to locally distinguish the different time bases. Most of the application

developers will only use one time base, so they can ignore this feature and go with the *Identity::k0. For those who are using/providing several time bases, e.g. two Synch SlaveTBs, can define their own type by casting any integer to the appropriate identity(e.g. SynchSlaveTB<static_cast<SynchSlaveIdentity>(1)>).