EB tresos classic AUTOSAR training
Software components and the RTE

Elektrobit

The chapter consists of 3 sections

Software Components

- AUTOSAR Software Component Description
- Basic Elements of an SWC
- How to create a SWC
- Connecting SWCs and Compositions
- Sender / Receiver interface
- Client / Server interface
- Other types of interfaces

The Runtime Environment

- The runtime environment (RTE)
- RTE generation Workflow
- RTE events and event mapping

Advanced SWC Concepts

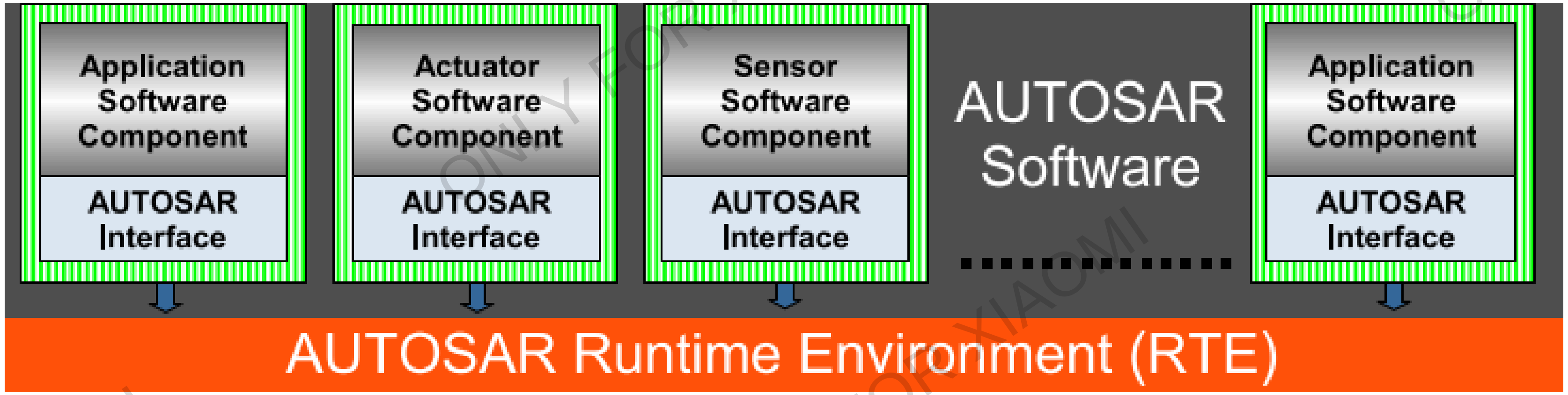
- Sender / Receiver
- Client / Server
- Interrunnable Variables
- Instantiation
- Exclusive areas
- Mode management
- Partitioning

AUTOSAR Software Component description

Elektrobit

The AUTOSAR Software component (SW-C)

- Software components (SW-C) are used to structure the application part of the system functionality
- Undividable SW-C are called **Atomic Software Components**
- AUTOSAR distinguishes between Application SW-C, Sensor SW-C and Actuator SW-C



Elektrobit

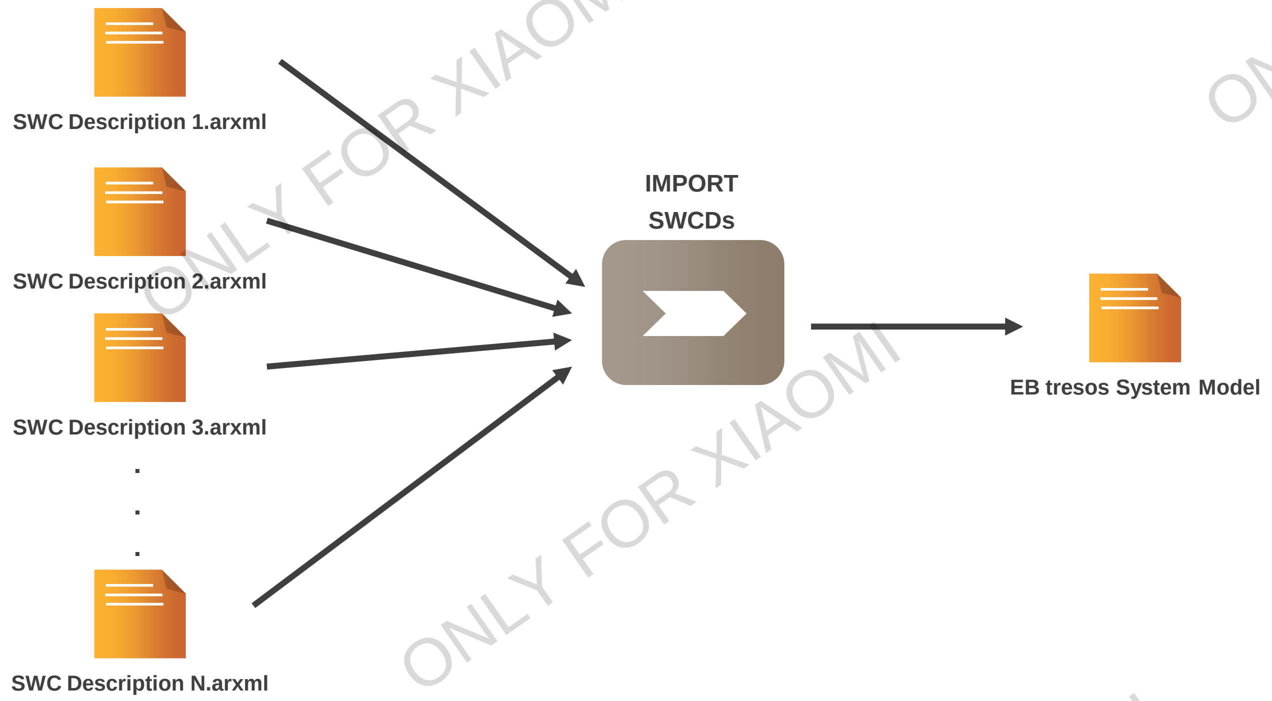
The Software component description (SWCD)

- The **SWC Description (SWCD)** contains all description elements to define a Software Component (SW-C), e.g.
  - Used interfaces (Operations and data elements provided and required)
  - Information regarding the specific implementation
- The formal language is defined by the AUTOSAR Standard and is called **AUTOSAR XML** (File extension .arxml)
- The SWCD file together with the SW-C source or object code is a complete **SW-C deliverable**

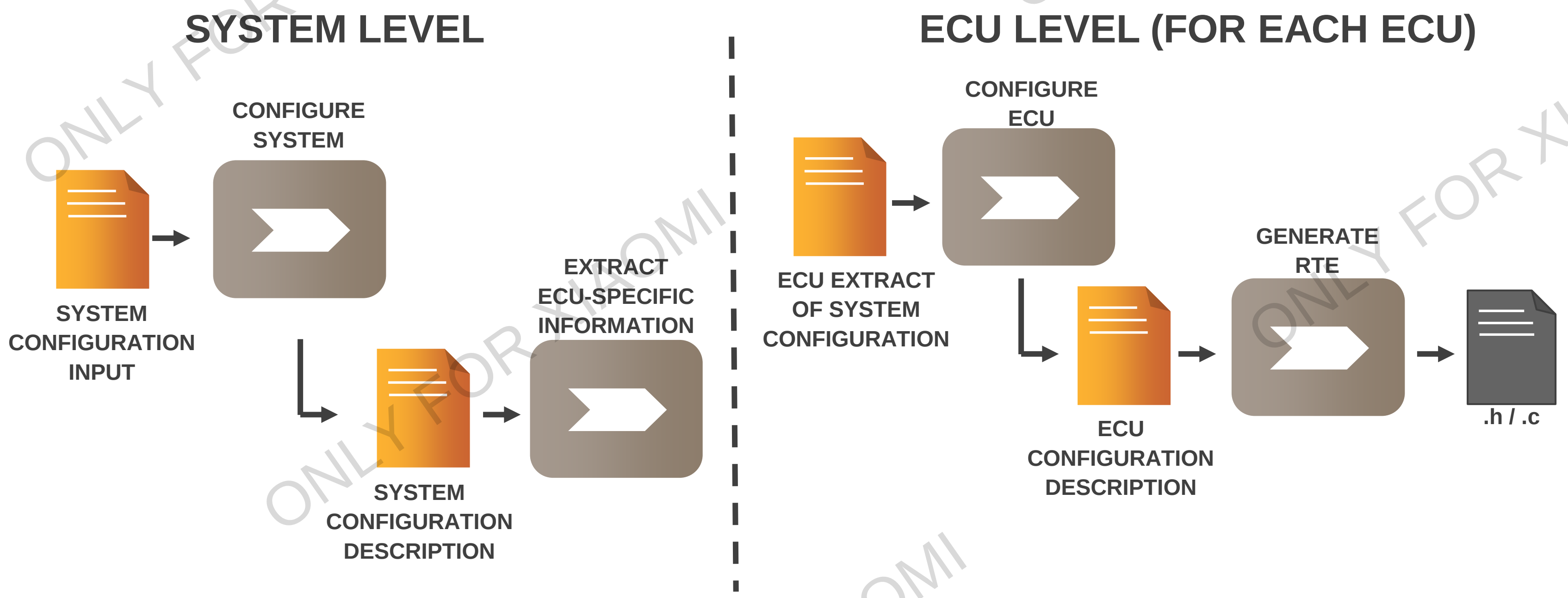
Elektrobit

SW-C Descriptions are imported as part of the System Description import

- The SWCD can be seen as a “plugin feature” of AUTOSAR
- All SWCDs are imported into the EB tresos internal System Model



…and will then be used to generate the RTE

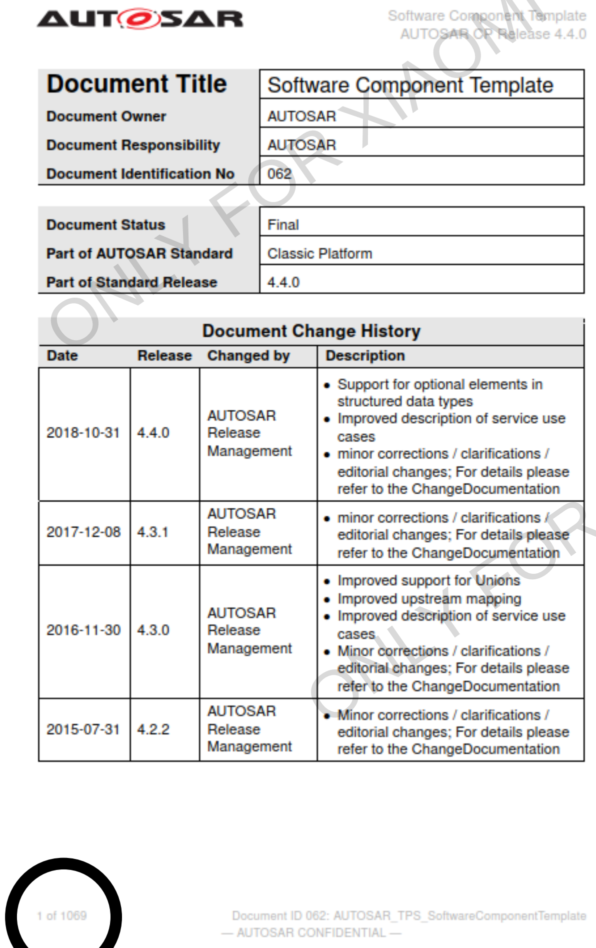


NOTE: *There is also a possibility to run the RTE generator on a standalone SWCD to get only the API header files - more about this later!*

Content of the SWCD

- AUTOSAR specifies the content and structure of the SWCD in the Software component Template
- A ***SWC Description*** (SWCD) consists of three main parts:
  - ***SWC Type Definition***
    - Defines all communication interfaces(APIs) that the SWC will need from the RTE
  - ***SWC Internal Behavior***
    - Defines runnables (functions), variables, events and other internal data used by the SWC
  - ***SWC Implementation***
    - Meta information such as used compiler, language and change log

> ***NOTE:*** *It can also contain auxiliary information such as DataTypes, Units and Calibration data definitions. However, the in the SWCD references to definitions are used which are defined on a project scope*


AUTOSAR

Software Component Template
AUTOSAR CP Release 4.4.0

Document Title	Software Component Template
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	062

Document Status	Final
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	4.4.0

Document Change History

Date	Release	Changed by	Description
2018-10-31	4.4.0	AUTOSAR Release Management	• Support for optional elements in structured data types • Improved description of service use cases • minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation
2017-12-08	4.3.1	AUTOSAR Release Management	• minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation
2016-11-30	4.3.0	AUTOSAR Release Management	• Improved support for Unions • Improved upstream mapping • Improved description of service use cases • Minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation
2015-07-31	4.2.2	AUTOSAR Release Management	• Minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation

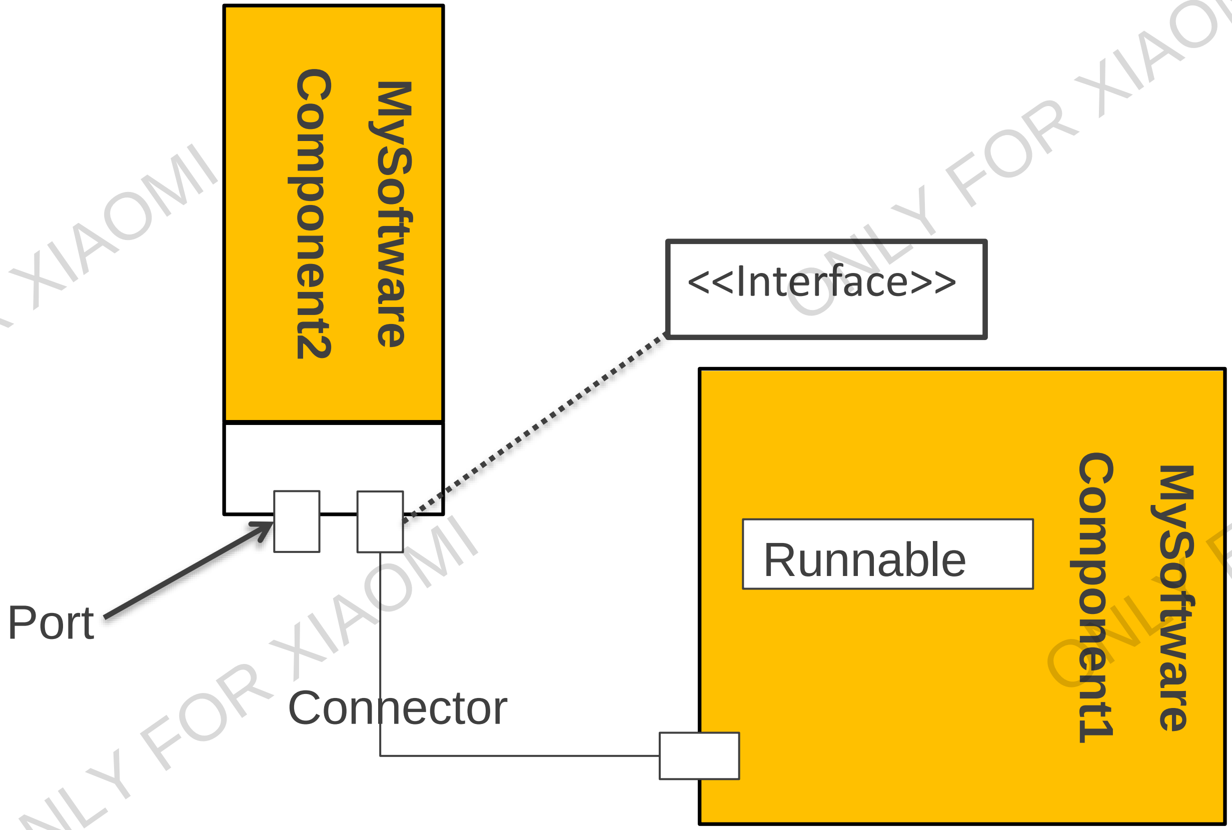
1 of 1069

Document ID 062: AUTOSAR_TPS_SoftwareComponentTemplate
— AUTOSAR CONFIDENTIAL —

A 1000+ page document

Basic Elements of an SW-C

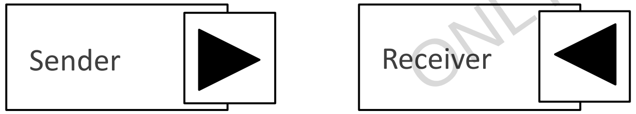
Basic Elements for SW-C - Overview



Basic Elements of an SW-C - Interfaces

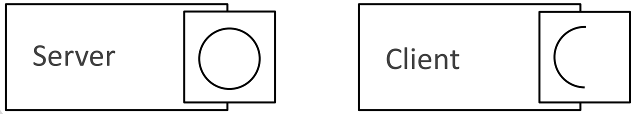
Semantics 1: *Sender/Receiver interface (S/R)*

- For broadcasting of signals (“DataElements”)
- One-way communication
- Many signals may be bundled in one S/R interface



Semantics 2: *Client/Server interface (C/S)*

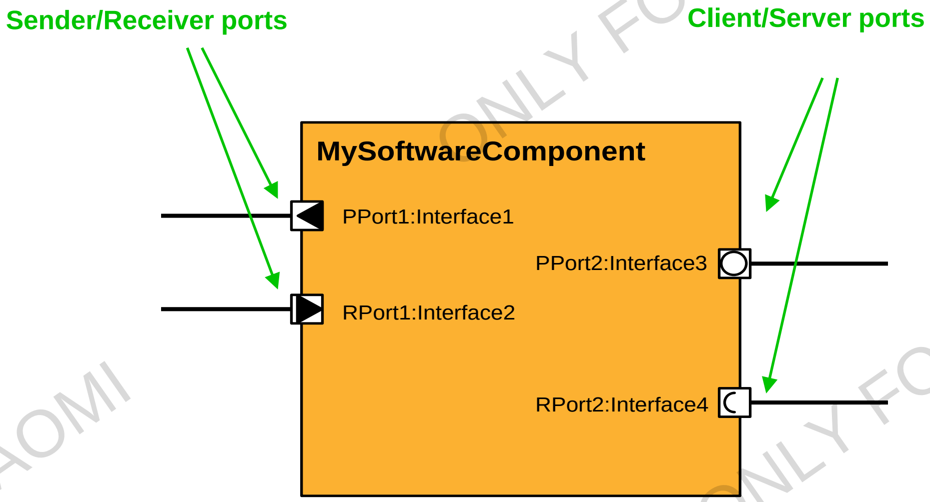
- For function invocations (with optional parameters and return value)
- Two-way communication (client waits for server to process request)
- Many functions may be bundled in one C/S interface



Interfaces define what kind of data that is transferred between SWCs as well as the semantics of the transfer

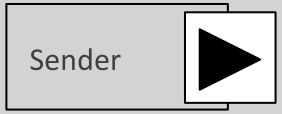
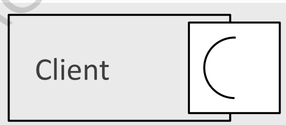
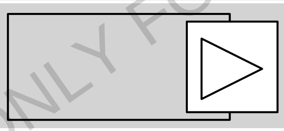
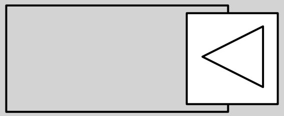
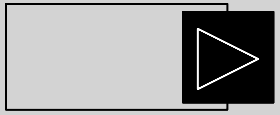
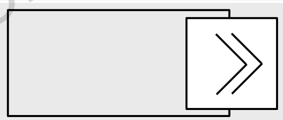
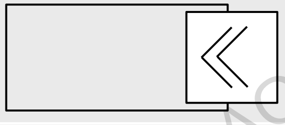
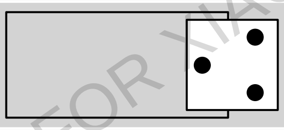
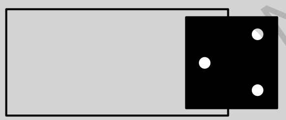
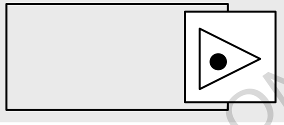
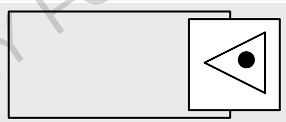
Basic Elements of an SW-C - Ports

- ***Ports*** are used by an SW-C to communicate with other SWCs
- Each port implements one Interface
  - AUTOSAR language: “The port is typed by the interface”



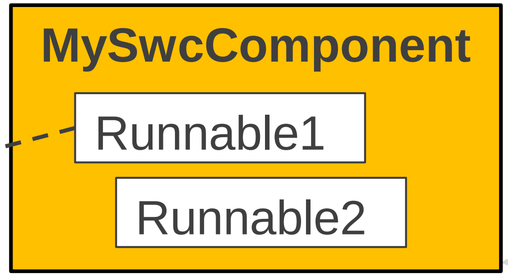
- ***Provide port (P-Port)***
  - S/R: A sender provides a signal (TX)
  - C/S: A server provides a service to a client
- ***Require port (R-Port)***
  - S/R: A receiver requires a signal (RX)
  - C/S: A client requires a service from a server
- ***Provide-require port (PR-Port)*** combine the ability to provide and require services or data in one entity

Port icons for different Interface types

Interface	Application Provide Port	Application Require Port	Service Provide Port	Service Require Port
Sender / Receiver				
Client / Server				
Parameter				
Trigger				
Mode Switch				
NV data				

Basic Elements of an SW-C - Runnable

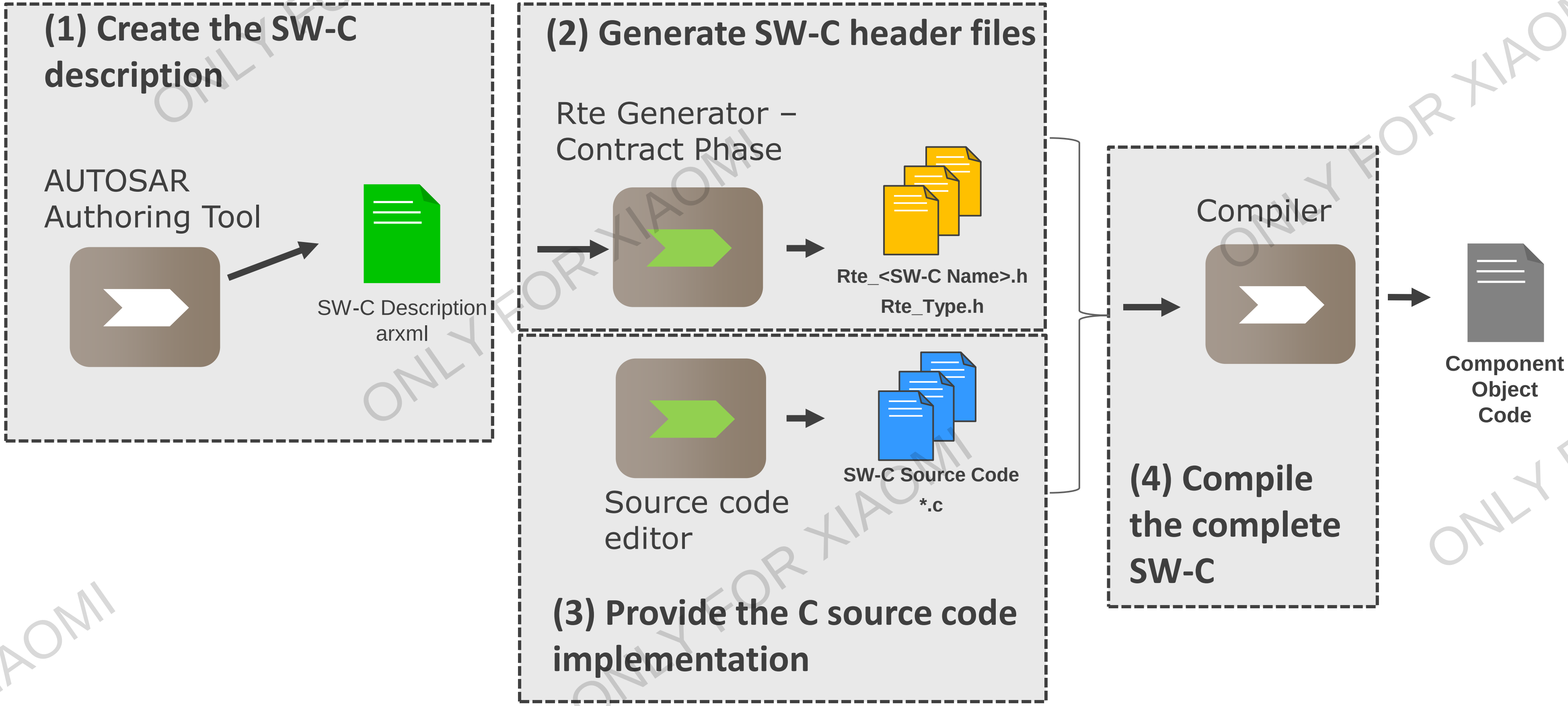
- A **runnable** is the AUTOSAR term for a function that is implemented as part of a SW-C
- A runnable can be time-triggered or event triggered (more about this later)
- One SW-C can have multiple runnables




```
/*MySwcComponent.c*/
void Runnable1 {
    /*code*/
}
```

How to create an SW-C

How to create an SW-C – Overview



Note: Contract phase is used to generate the SW-C header files based on a partial AUTOSAR system (SW-C description)

(1) Create the SW-C description

The SW-C Description

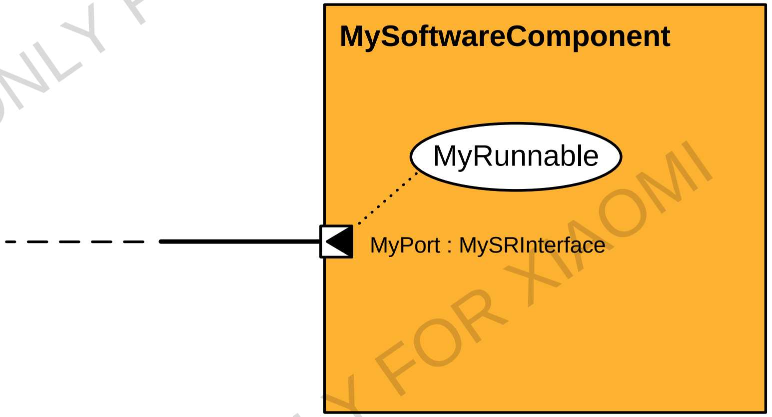
- Tools used to define SWCDs are called „AUTOSAR Authoring tools“
- EB tresos Studio is not an AUTOSAR Authoring tool but EB cooperates with all major tool vendors to ensure interoperability
- In the following examples we will use Artext - a textual representation of an AUTOSAR Model
- Note: Artext is not a commercial tool and only supports a subset of the AUTOSAR standard

Examples in Artext will be highlighted with this green rectangle

(1) Create the SW-C description

Example SW-C: “MySoftwareComponent”

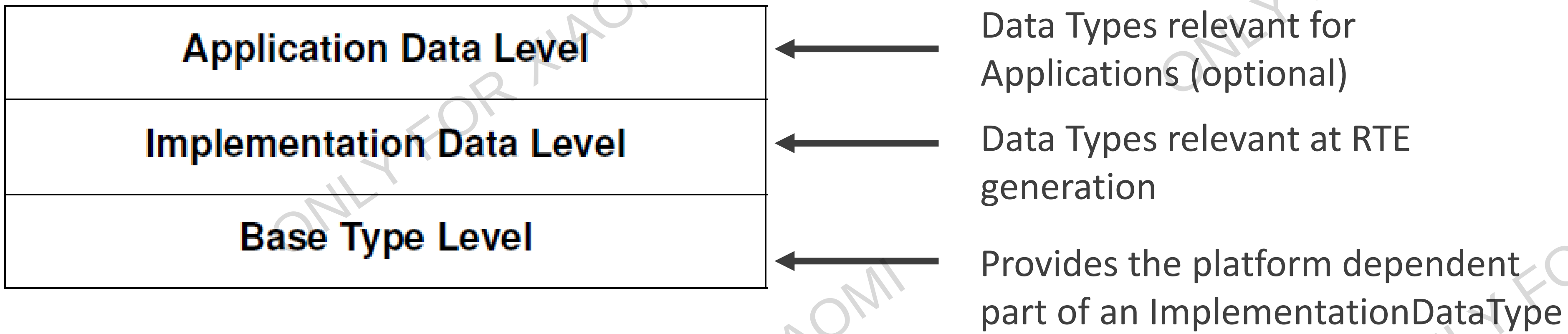
- Properties of our SW-C:
  - 1 ***Interface*** defining a simple 8-bit unsigned integer signal → Let’s call it “MySRInterface”
  - 1 ***Port*** providing a data element as defined in the Interface → Let’s call it “MyPort”
  - 1 ***Runnable*** which shall be called every 1000ms that can access our port → Let’s call it “MyRunnable”



(1) Create the SW-C description

Background information – Data Types

- AUTOSAR defines three different levels of abstraction



(1) Create the SW-C description

Background information – Data Types

- ***Data types*** can be defined from a number of base data types

- ***Primitive data types, which allow a direct mapping to C intrinsic types, e.g.*** :
  - BooleanType
  - IntegerTypes (up to 64 bits)
  - FloatType
  - Opaque (bit field)

- ***Complex (composite) data types which map to C arrays and structures.***:
  - ArrayDataType (arrays)
  - RecordDataType (structs)

(1) Create the SW-C description

Background information – Data Types

- Usually, the OEM defines a number of DataTypes for the AUTOSAR system
- If you create project specific DataTypes, it is a good strategy to…
  - …add your AUTOSAR DataTypes in a common place (a "package" or "library")
  - …have a naming strategy for DataTypes since they will eventually turn up in the source code

(1) Create the SW-C description

Example of a Data Type description

- We configure a simple 8-bit unsigned integer:

int myUInt8 **min** 0 **max** 255 **extends** uint8

```
<IMPLEMENTATION-DATA-TYPE>
  <SHORT-NAME>myUInt8</SHORT-NAME>
  <CATEGORY>VALUE</CATEGORY>
  <SW-DATA-DEF-PROPS>
    <SW-DATA-DEF-PROPS-VARIANTS>
      <SW-DATA-DEF-PROPS-CONDITIONAL>
        <BASE-TYPE-REF DEST="SW-BASE-TYPE">/AUTOSAR_Platform/BaseTypes/uint8</BASE-TYPE-REF>
        <DATA-CONSTR-REF DEST="DATA-CONSTR">/AUTOSAR_Platform/DataConstrs/uint8</DATA-CONSTR-REF>
      </SW-DATA-DEF-PROPS-CONDITIONAL>
    </SW-DATA-DEF-PROPS-VARIANTS>
  </SW-DATA-DEF-PROPS>
  <TYPE-EMITTER>BSW</TYPE-EMITTER>
</IMPLEMENTATION-DATA-TYPE>
```

(1) Create the SW-C description

Example of a Base Type and Data Constraint definition

```
<SW-BASE-TYPE>
  <SHORT-NAME>uint8</SHORT-NAME>
  <CATEGORY>FIXED_LENGTH</CATEGORY>
  <BASE-TYPE-SIZE>8</BASE-TYPE-SIZE>
  <BASE-TYPE-ENCODING>NONE</BASE-TYPE-ENCODING>
  <MEM-ALIGNMENT>8</MEM-ALIGNMENT>
  <NATIVE-DECLARATION>unsigned char</NATIVE-DECLARATION>
</SW-BASE-TYPE>
```

```
<DATA-CONSTR>
  <SHORT-NAME>uint8</SHORT-NAME>
  <DATA-CONSTR-RULES>
    <DATA-CONSTR-RULE>
      <INTERNAL-CONSTRS>
        <LOWER-LIMIT INTERVAL-TYPE="CLOSED">0</LOWER-LIMIT>
        <UPPER-LIMIT INTERVAL-TYPE="CLOSED">255</UPPER-LIMIT>
      </INTERNAL-CONSTRS>
    </DATA-CONSTR-RULE>
  </DATA-CONSTR-RULES>
</DATA-CONSTR>
```

(1) Create the SW-C description

Interface description

- We use our DataType by defining a ***DataElement*** in a ***SenderReceiverInterface***

```
interface senderReceiver  MySRInterface {
        data myUInt8 myFirstAutosarSignal
}
```

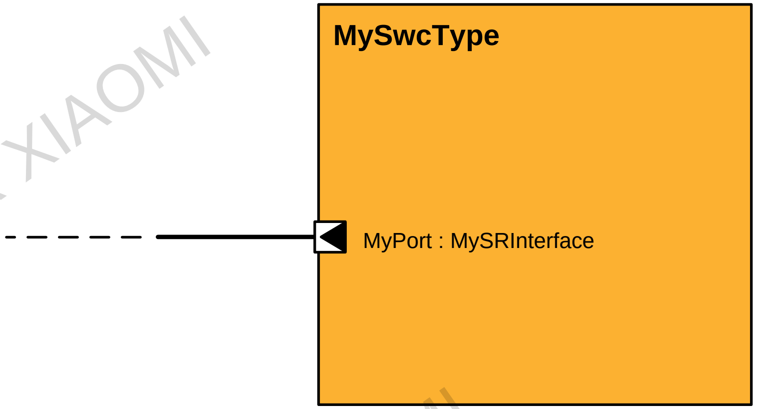
- The interface is a contract between two (or more) SWCs on how they shall communicate

(1) Create the SW-C description

SW-C type and Port

- We package our SenderReceiverInterface into a ***Port*** in a ***SW-C Type***

```
component application MySwcType {
    ports {
      sender MyPort provides MySRInterface
    }
}
```



Elektrobit

(1) Create the SW-C description

SW-C Description - Completeness check

- A ***SWC Description*** (SWCD) consists of three main parts:
  - ***SWC Type Definition***
    - Defines all communication interfaces(APIs) that the SWC will need from the RTE
  - ***SWC Internal Behavior***
    - Defines functions, variables, events and other internal data needed by the SWC
  - ***SWC Implementation***
    - Meta information such as used compiler, language and change log

(1) Create the SW-C description

Create the Internal Behavior description

- The SWC consists of one ***Runnable*** (function)
- The ***Runnable*** should be scheduled every 1000 ms

```
internalBehavior mySwcIB for MySwcType {
    runnable MyRunnable [1.0] {
        /* Port: /arpRoot/MySwcType/MyPort */
        /* DataElement: /arpRoot/MySRInterface/myFirstAutosarSignal */
        dataSendPoint MyPort.myFirstAutosarSignal

        /* Type: /arpRoot/MySwcType */
        /* Periode: 1000 ms  */
        /* Runnable: /arpRoot/mySwcIB/MyRunnable */
        timingEvent 1.0
    }
}
```

The **DataSendPoint** is used to define that the Runnable shall be able to access the Port

(1) Create the SW-C description

Essential parts of the SW-C Description - Completeness check

- A ***SWC Description*** (SWCD) consists of three main parts:
  - ***SWC Type Definition***
    - Defines all communication interfaces(APIs) that the SWC will need from the RTE
  - ***SWC Internal Behavior***
    - Defines functions, variables, events and other internal data needed by the SWC
  - ***SWC Implementation***
    - Meta information such as used compiler, language and change log

(1) Create the SW-C description

Add the SW-C Implementation description

- The ***Implementation*** is just some meta information that can be used the toolchain

```
Implementation myImpl for mySwcIB {
    language C
    codeDescriptor "src“

}
```

- The only mandatory information in the Implementation description is the reference to the InternalBehavior

(1) Create the SW-C description

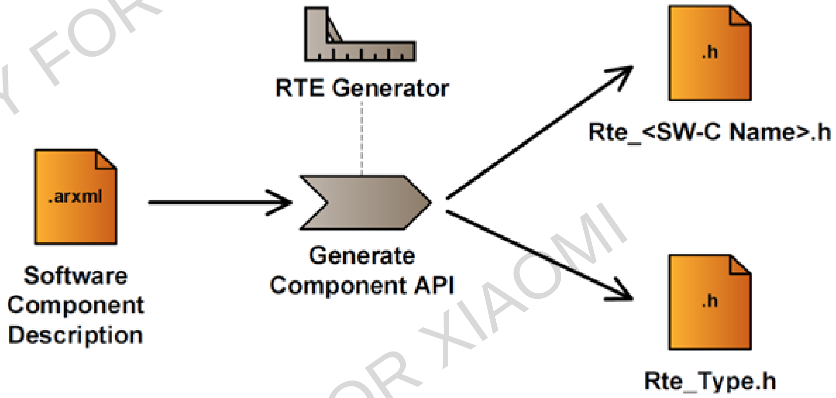
Parts of the SW-C Description - Completeness check

- A ***SWC Description*** (SWCD) consists of three main parts:
  - ***SWC Type Definition***
    - Defines all communication interfaces(APIs) that the SWC will need from the RTE
  - ***SWC Internal Behavior***
    - Defines functions, variables, events and other internal data needed by the SWC
  - ***SWC Implementation***
    - Meta information such as used compiler, language and change log

(2) Generate SW-C header files

Run the RTE generator in “contract phase”

- The SW-C description is now complete and may be exported by the AUTOSAR Authoring tool to an ARXML file
- This ARXML file is then used by the RTE generator in ***Contract phase*** *to create the required header files*



(2) Generate SW-C header files

- The RTE generator will produce ***Rte_MySwcType.h*** and ***Rte_Type.h***:

Rte_MySwcType.h *(simplified):*

```
#ifndef _RTE_MYSWCTYPE_H
#define _RTE_MYSWCTYPE_H

#include "Rte_Type.h"

/* Runnable functions */
void MyRunnable(void);

/* Communication APIs */
 Std_ReturnType Rte_Write_MyPort_myFirstAutosarSignal(myUInt8 data);
#endif /* _ RTE_MYSWCTYPE_H */
```

Rte_Type.h is generated by the RTE generator

This function must be implemented by us

The RTE will implement this function

This data type is declared in Rte_Type.h

Rte_Type.h *(simplified):*

```
#ifndef _RTE_TYPE_H
#define _RTE_TYPE_H
typedef uint8 myUInt8;
#endif /* _ RTE_TYPE_H */
```

(3) - Implement SW-C source code

- The implementation source code for our SW-C could look like this:
- ***Rte_MySwcType.c*** *(example):*

```
/* Include RTE APIs */
#include "Rte_MySwcType.h”
/* Runnable “MyRunnable" will be scheduled by the RTE every 1000 ms */
 void MyRunnable(void)
 {
    /* Signal variable */
    static myUInt8 myVar = 0;
    /* Send DataElement “myFirstAutosarSignal" via Port “MyPort" */
   Rte_Write_MyPort_myFirstAutosarSignal(myVar);
  myVar++;
 }
```

(4) - *Compile*

- Now all parts are ready and our SW-C can be compiled and sent to the integrator (either as source code or as object code).
- The integrator will include the SW-C description in the ECU Configuration Description
- The integrator’s RTE generator will generate exactly the same APIs as ours did when we ran it in contract phase (hence “contract”)

Elektrobit

SOFTWARE COMPONENTS - *Summary*

- A SW-C consists of a ***SW-C description*** and ***source code*** (static and generated). The ***RTE Generator*** provides the generated code part based on the SW-C description.
- AUTOSAR ***data types*** are used to build up ***interfaces*** which are implemented in ***ports***
- A ***sender/receiver*** interface is used for <u>one-way communication</u> while a ***client/server*** interface is used for <u>function calls</u>
- A ***runnable*** is a schedulable function in the SW-C that can access the ports via ***access points***. The RTE will provide the runnable with the APIs

Connecting SWCs and Compositions

Elektrobit

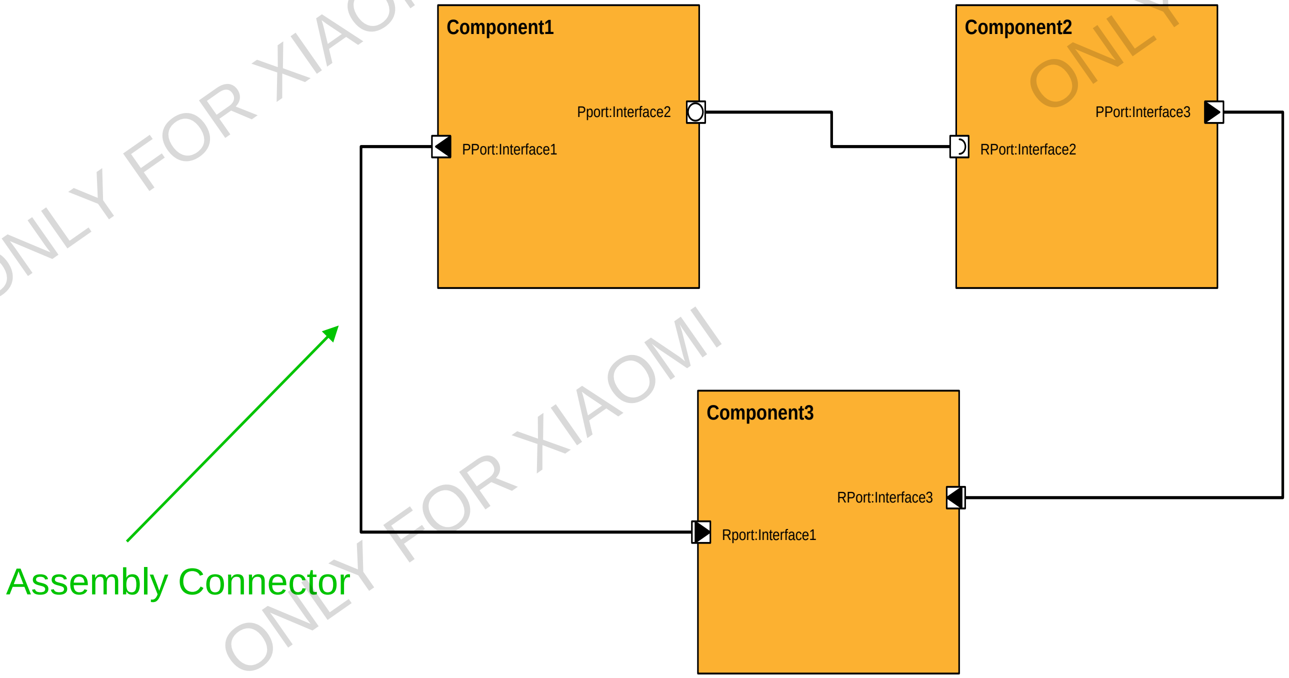
Elektrobit

Assembly Connectors

- SWCs are interconnected with ***AssemblyConnectors***

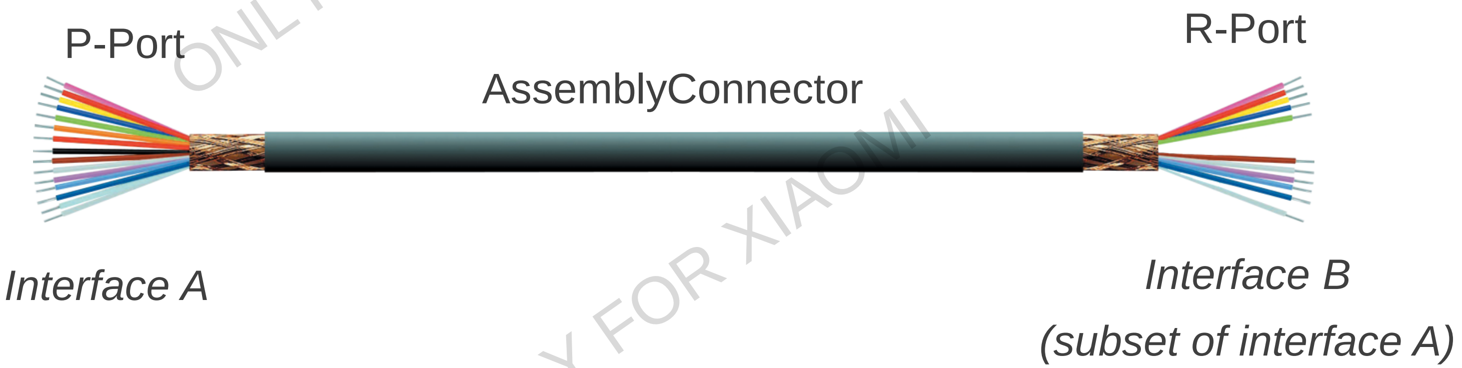
  - An AssemblyConnector defines a data binding between one PPort and one RPort allowing them to communicate

  - One port can have many AssemblyConnectors



Assembly Connectors - Interface compatibility

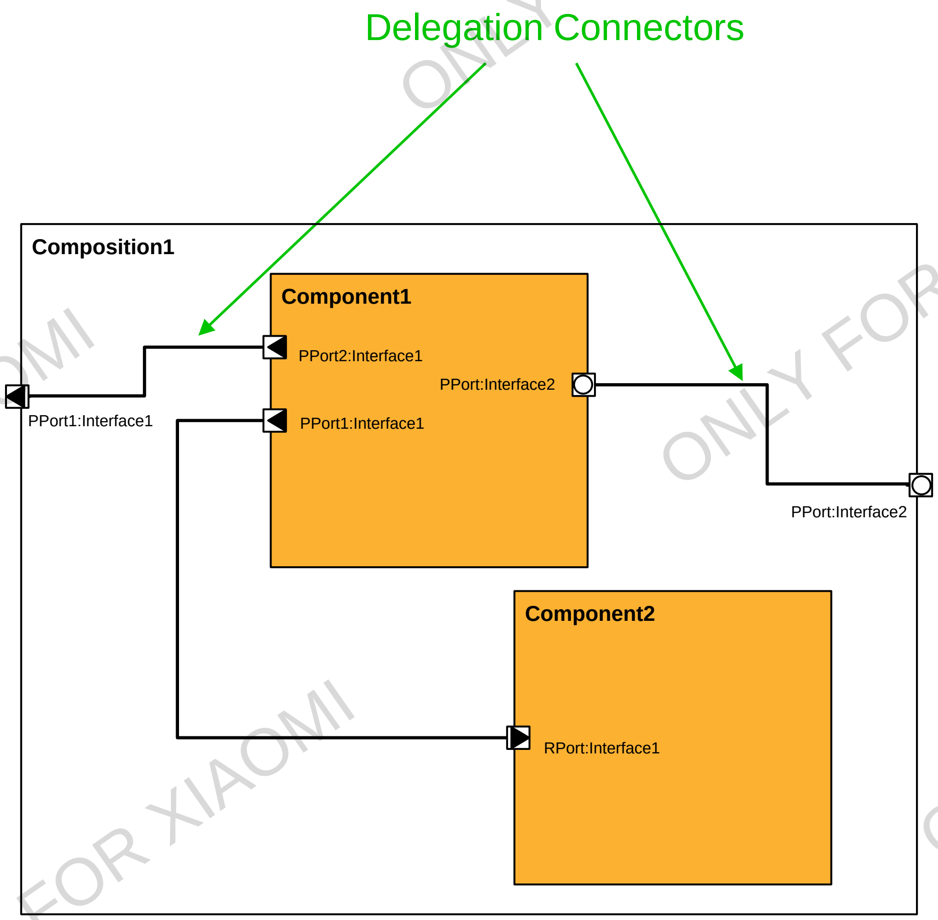
- A PPort may only be connected to an RPort as long as their interfaces are ***compatible:***
  - Same type of interface (sender/receiver vs. client/server)
  - Same signal/function names and data types
- The RPort's interface may be a subset of PPort's interface



Elektrobit

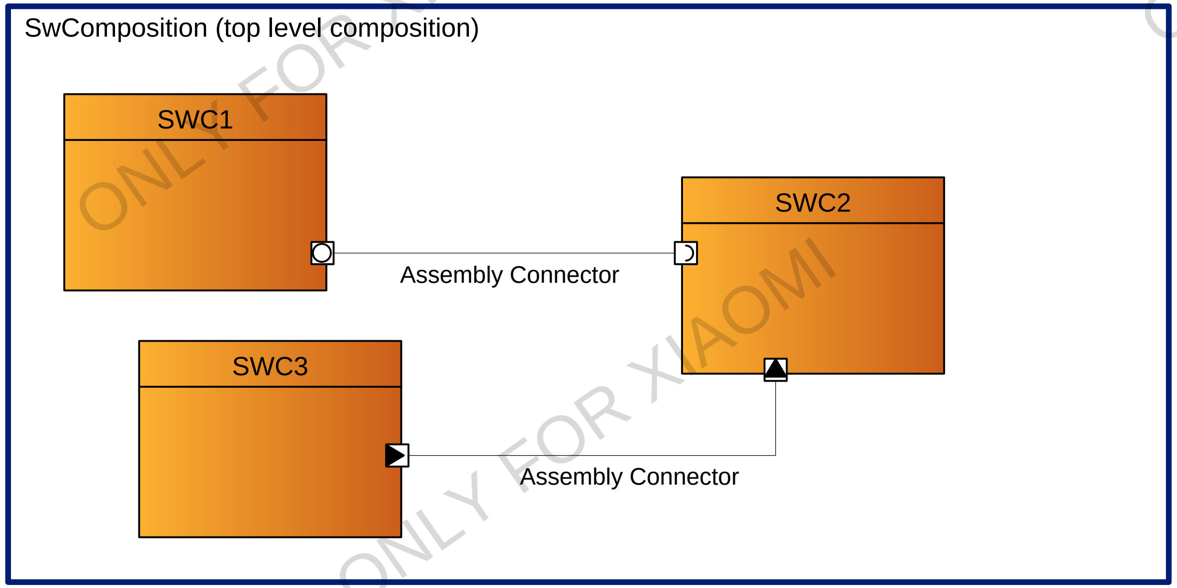
Compositions

- Several SWCs may be nested into a ***Composition***
- A composition will act as a SWC, hiding the content from the outside world (“black box”)
- ***Delegation Connectors*** are used to extend/expose an inner port to the outside
- Compositions allow you to
  - Group SWCs logically
  - Define interfaces between different suppliers
  - Re-arrange the content of a composition without influencing the rest of the system



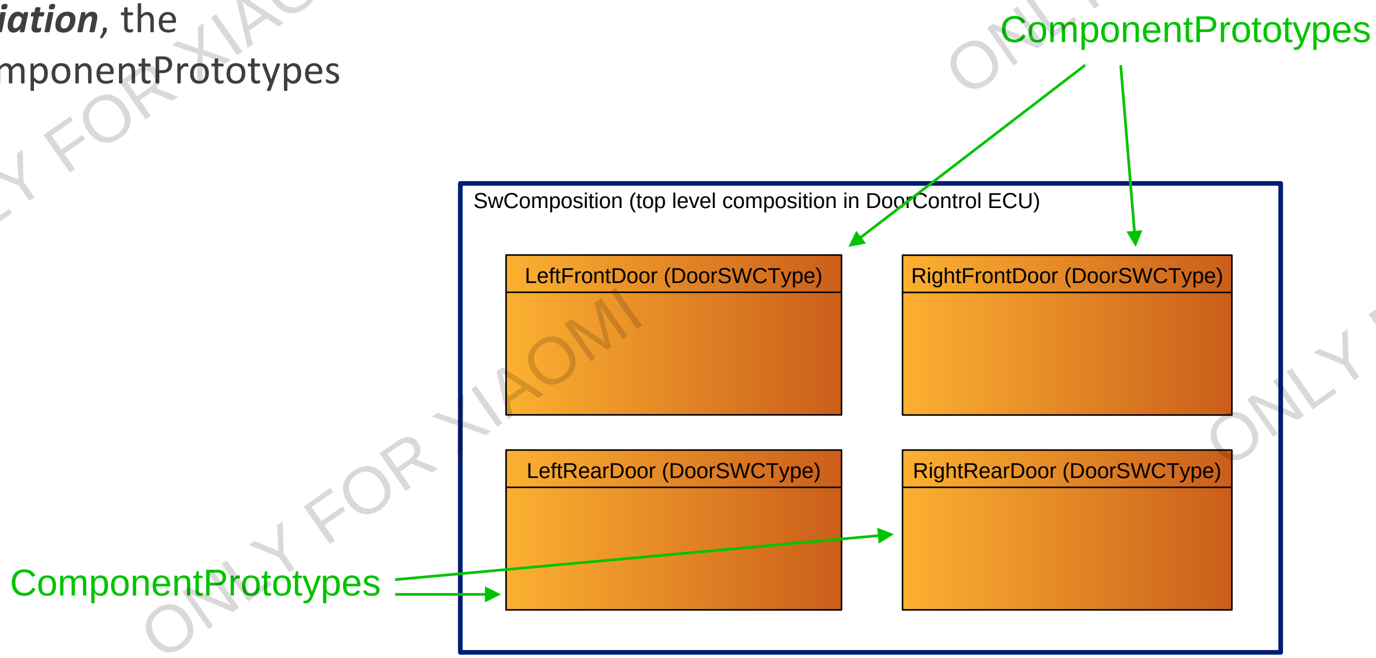
SW Composition - *The Top Level Composition*

- In an AUTOSAR system, there is always at least one composition – the ***SwComposition*** (also called the "top level composition")
- It is the top-most composition, holding all other SWCs and Compositions



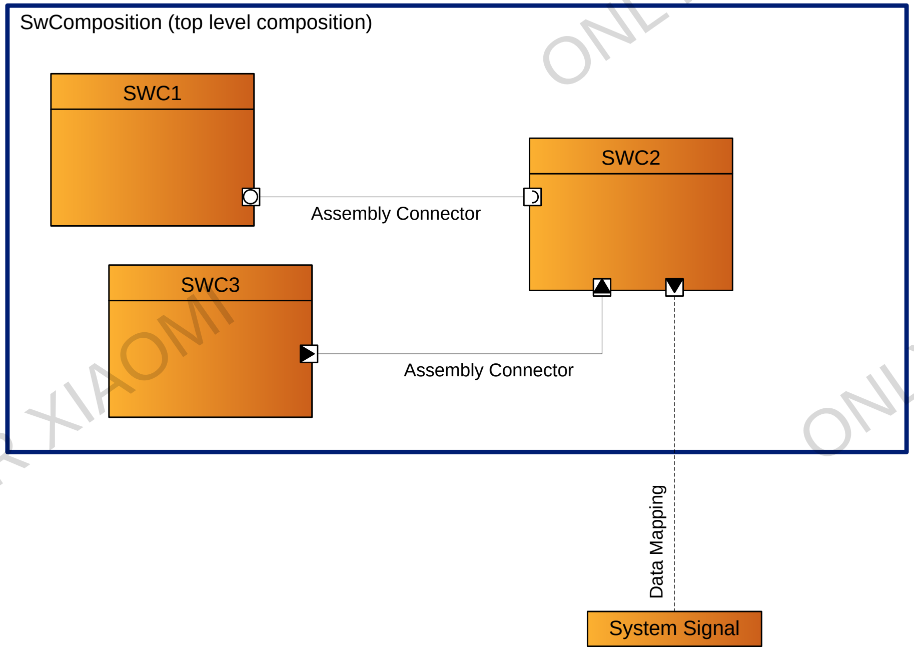
SW-C Instantiation

- A component ***instance*** in a composition is called a ***ComponentPrototype***
- The ComponentPrototype references the SWC Type
- If the SWC Type supports ***multiple instantiation***, the same type can be instantiated in many ComponentPrototypes
- ***Example:*** Door lock application – the same SWC can be re-used for all four doors in a car by instantiating it four times in different Component-Prototypes



Data Mappings - Communication outside the ECU

- AssemblyConnectors only allow communication between SWCs in the same ECU (“intra-ECU”)
- To communicate with other ECUs (“inter-ECU”), we must use the communication stack
- A ***DataMapping*** is used to map a signal in the communication stack to a DataElement in a certain port
- Since one port can contain many DataElements (signals), there may be many DataMappings needed for the same port



Sender / Receiver interface

Elektrobit

Elektrobit

Sender / Receiver interface

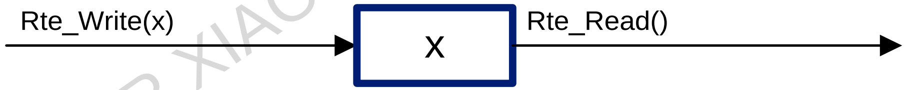
- An S/R interface may contain one or more ***Data Elements*** (signals)
- A Data Element always references a ***DataType***

```
interface senderReceiver
mySRInterface
{
  data myUInt8 elem1

  data mySInt8 elem2

  data myUInt32 elem3
}
```

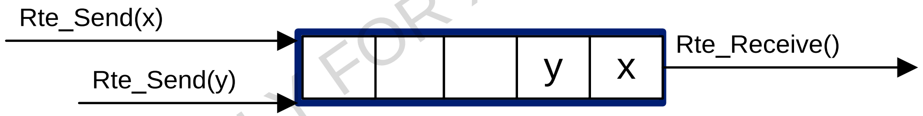
Sender / Receiver – queued and unqueued

A Data Element can be defined in two semantics

- Unqueued (“last-is-best” semantics):
  - The receiver will only read the most recently received value



- Queued (“event” semantics):
  - The receiver will have a FIFO queue of configurable length to be able to receive values in the order they arrive
  - The RTE provides the queue functionality to the SWC



Elektrobit

Sender/Receiver Interfaces - Unqueued API

- Example: S/R transmission of an unqueued data element
  (SWC implementation code)

```
Rte_Write_myPPort_myDataElement(myVar);
```

- Example: S/R reception of an unqueued data element
  (SWC implementation code)

```
myUInt16 myVar;
Rte_Read_myRPort_myDataElement(&myVar);
```

Sender/Receiver Interfaces - Queued API

- Example: S/R transmission of a <u>queued</u> data element
  (SWC implementation code)

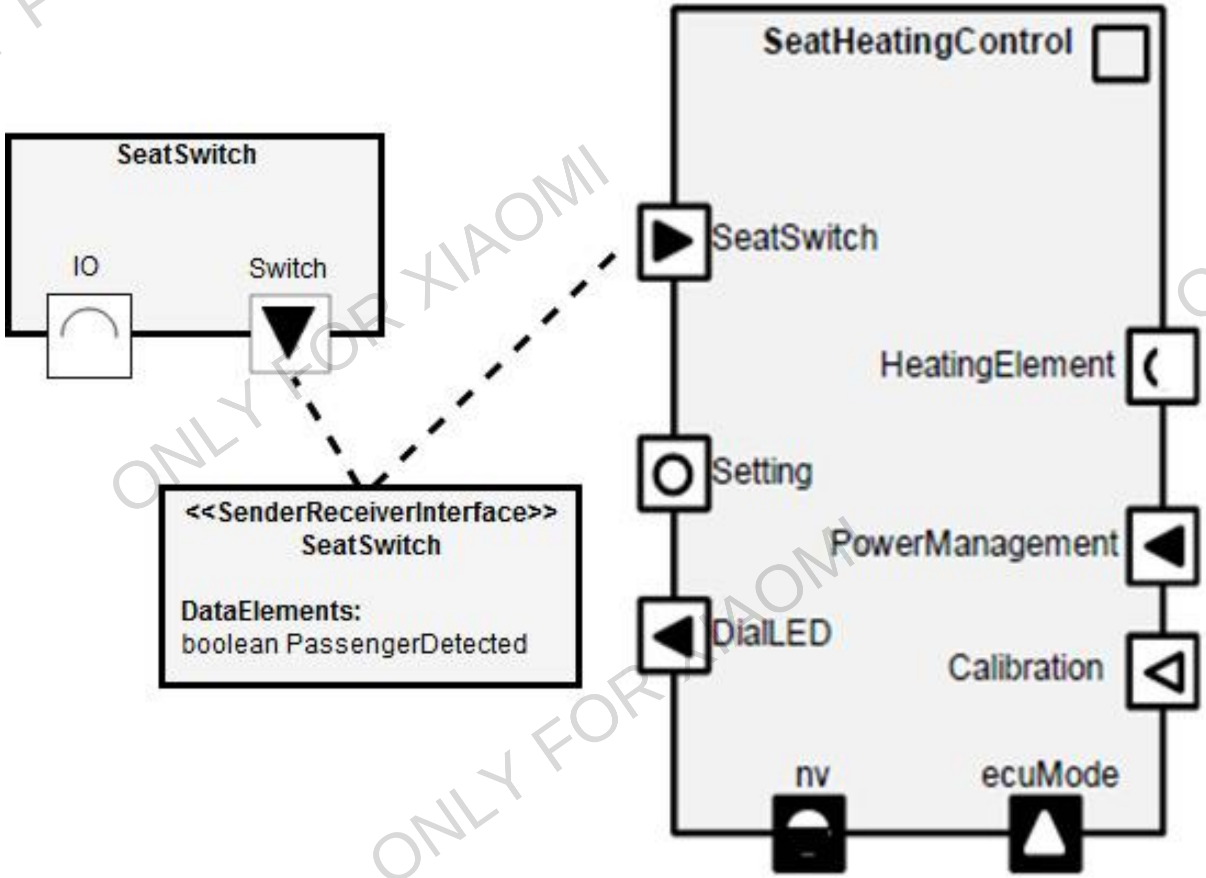
```
Rte_Send_myPPort_myDataElement(myVar);
```

- Example: S/R reception of a <u>queued</u> data element
  (SWC implementation code)

```
myUInt16 myVar;
while (Rte_Receive_myRPort_myDataElement(&myVar) == RTE_E_OK)
{
        /* Do something with the data read from queue */
}
```

Sender/Receiver Interfaces – Usecases

- Concrete example:



Sender/Receiver - Interface compatibility

- Two S/R interfaces are compatible if…
  - All the required DataElements in the R-Port are provided in the connected P-Port (i.e. the DataElements in the R-Port is a subset of the DataElements in the P-Port)
  - The DataTypes of the DataElements are compatible
  - The names of the DataElements are identical

Client / Server interface

Client/Server Interfaces

- A C/S interface may contain one or more ***Operations*** (functions)
- Each operation contains zero or more ***Arguments***
  - Direction may be “IN”, “OUT” or “IN/OUT”
- Each operation contains zero or more ***Error Return Codes***

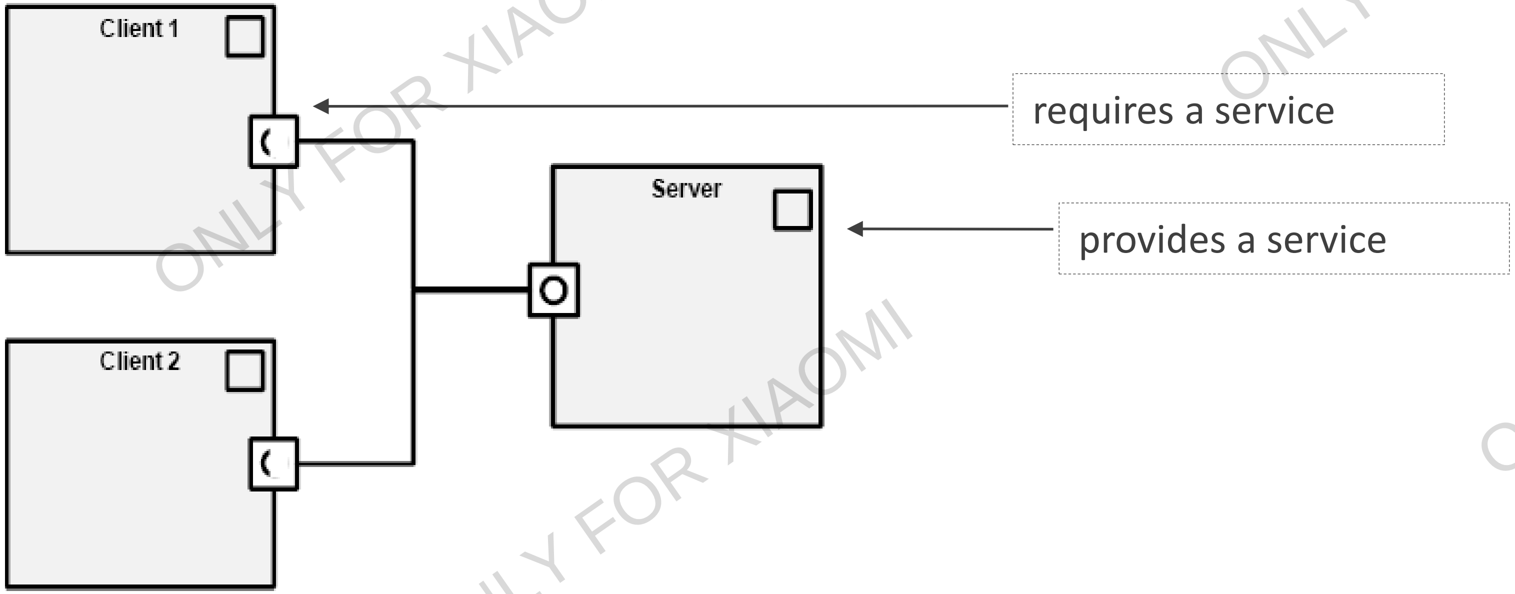
> Error code “0” is always reserved by RTE_E_OK

```
interface clientServer myClientServerInterface
{
    error myError1 1
    error myError2 2

    operation myOperation1 possibleErrors
     myError1, myError2
    {
        in myBoolean myArg1

        out myUInt8 myArg2

        inout myUInt16 myArg3
    }

    operation myOperation2
    {
        …
    }

}
```

Client/Server Interfaces - *PPorts and RPorts*

- A Server *provides* a service via a PPort

- The clients may invoke the server by having their RPorts connected to the server port via AssemblyConnectors (the client *requires* a service)



Client/Server Interfaces - Example

- ***Example***: Client invocation of an operation with one IN argument, one OUT argument and error codes enabled

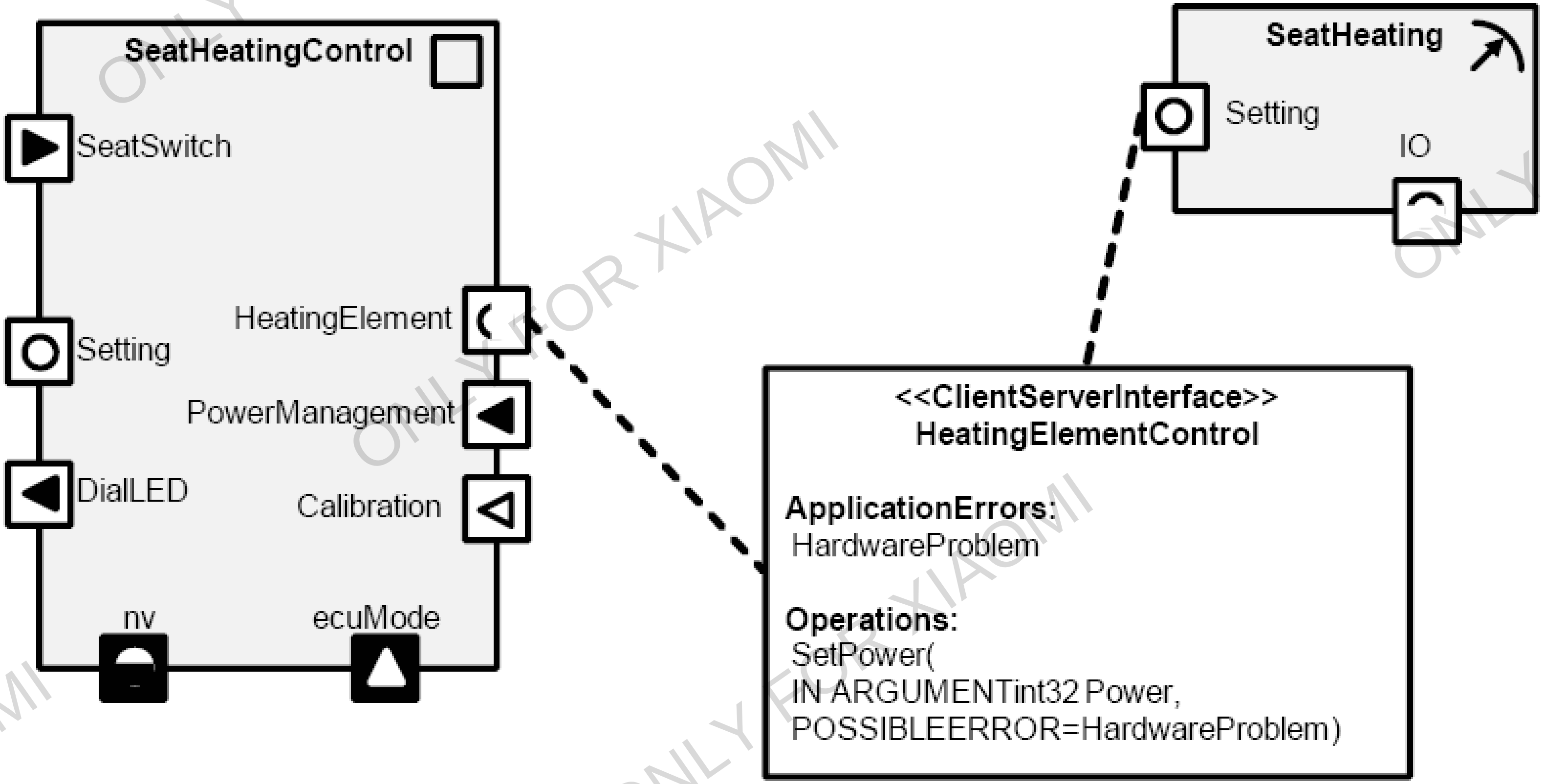
```
UInt32 myOutArg;
Std_ReturnType myErr;
myErr = Rte_Call_myPPort_myOperation(myInArg, &myOutArg);
if (myErr != RTE_E_OK)
{
       /* Error handling here... */
}
```

- ***Example***: Corresponding server implementation

```
Std_ReturnType myOperation(UInt8 myInArg, UInt32* myOutArg)
{
       /* Server code here... */
}
```

Client/Server Interfaces – Usecases

- Concrete example:



Client/Server - Interface compatibility

- Two C/S interfaces are compatible if…
  - All the required Operations in the RPort are provided in the connected server PPort
  - The names of the Operations are identical
  - The Arguments are compatible (same name, same direction, data types compatible)

Other types of interfaces

Elektrobit

Port icons for different kind of interfaces

Interface	Application Provide Port	Application Require Port	Service Provide Port	Service Require Port
Sender / Receiver	Sender	Receiver		
Client / Server	Server	Client		
Parameter				
Trigger				
Mode Switch				
NV data				

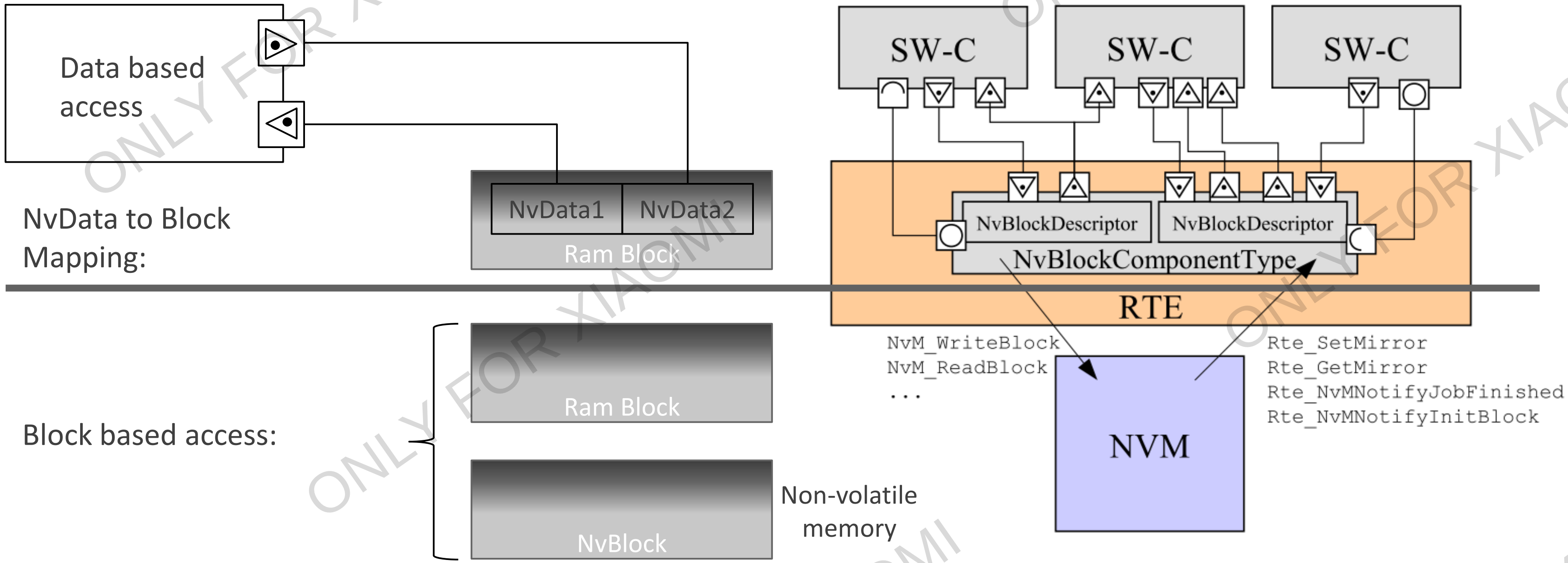
Other Interfaces

- Trigger Interface
  - Used to trigger execution of a runnable
  - like Sender/Receiver communication without data but with date receive event
  - Additionally, a direct call (execution in context of caller is possible)

- Parameter Interface
  - Instead of a dataElement (VariableDataPrototype), a ParameterDataPrototype is used which is used to read Constants. These Constants can be modified via XCP.

- Mode Switch Interface
  - Instead of a dataElement (VariableDataPrototype), a ModeDeclarationGroupPrototype is used which enumerates all states

Data Interfaces (NV data)



- NvBlockComponentType is generated by RTE
  - Offers data based access for SW-Cs
  - Block based access additionally possible

Section Summary - Software Components (SWC)

- AUTOSAR Software Component Description
- Basic Elements of an SWC
- How to create a SWC
- Connecting SWCs and Compositions
- Sender / Receiver interface
- Client / Server interface
- Other types of interfaces

The runtime environment (RTE)

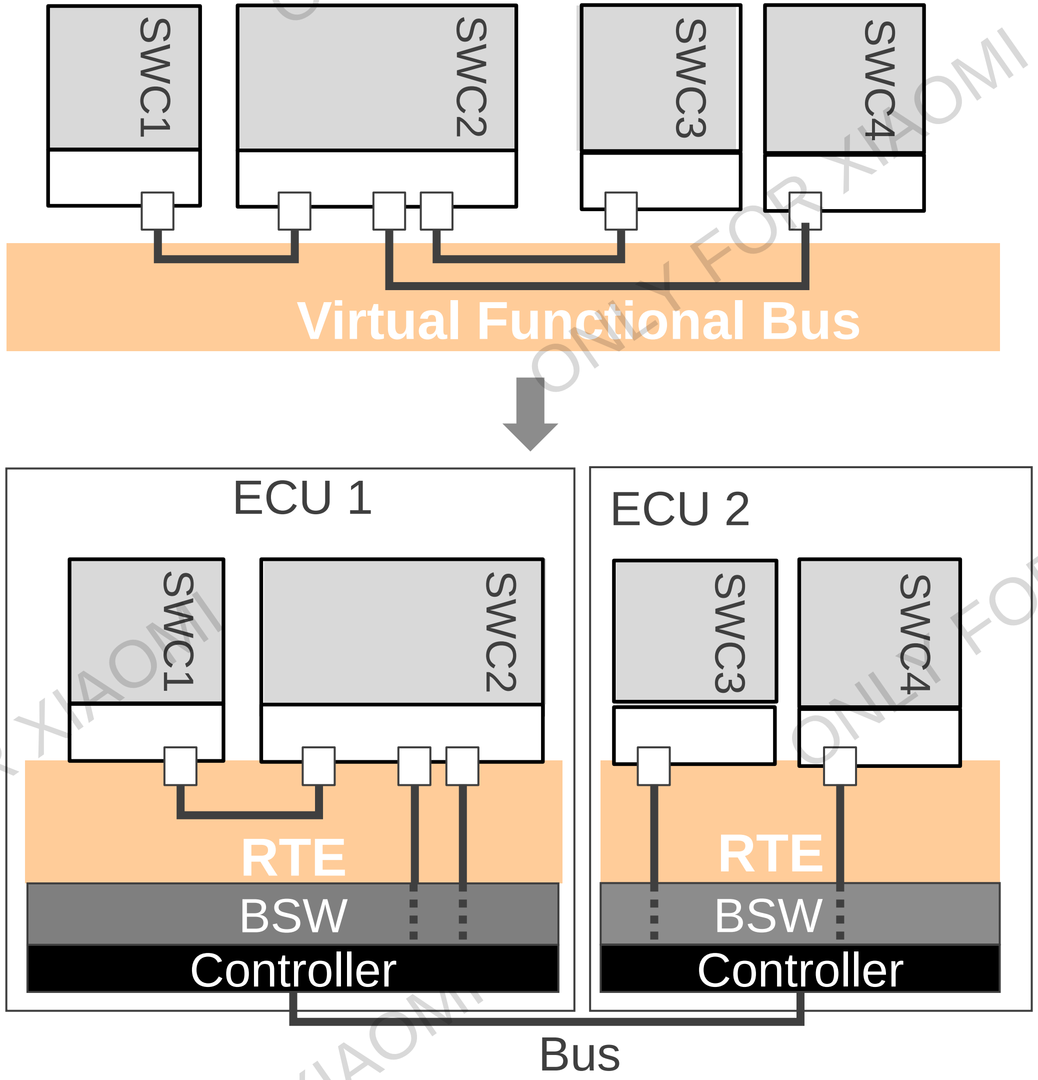
Section overview

- The runtime environment (RTE)
- RTE generation Workflow
- RTE events and event mapping

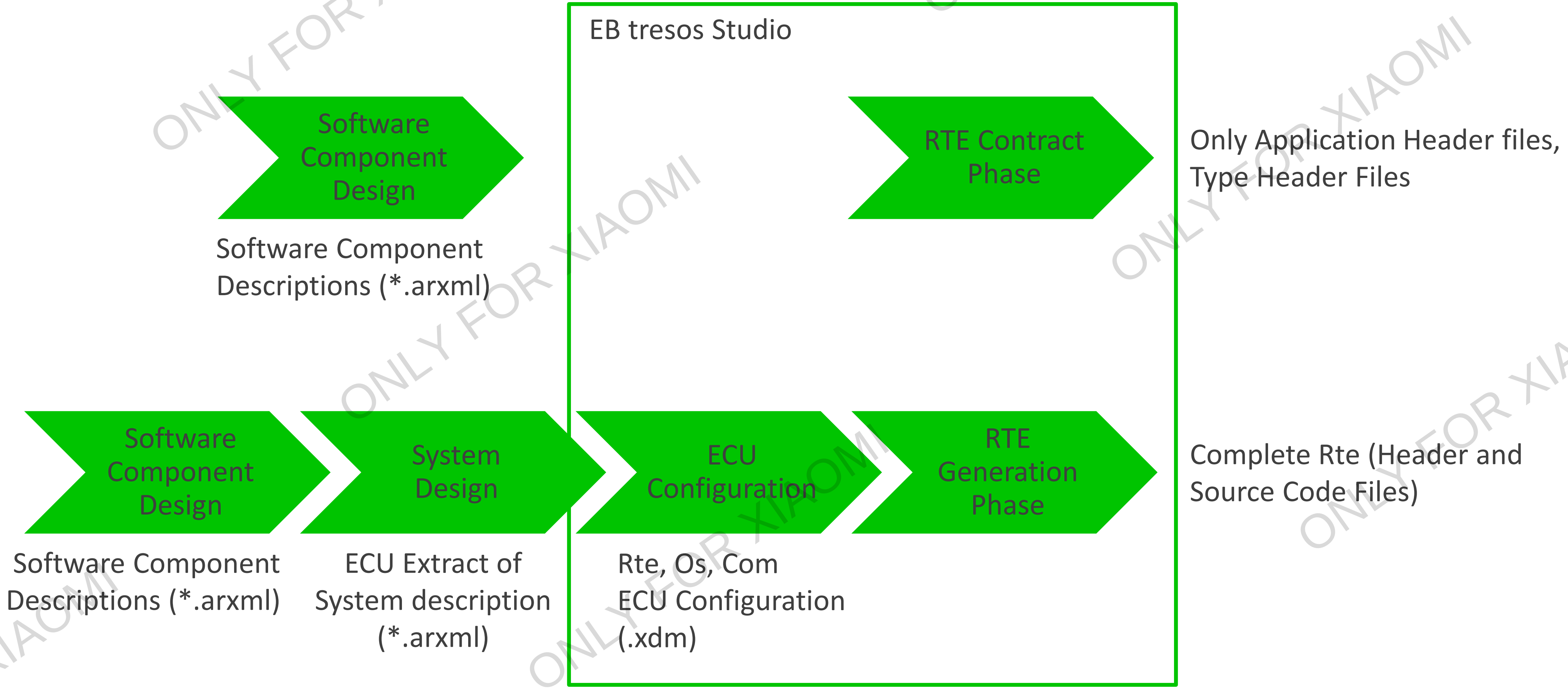
The Runtime environment (RTE)

Quick review...

- The ***RTE*** implements the ***Virtual Functional Bus*** for one ECU
- It allows SW-Cs to use the same communication APIs no matter if the destination SW-C is on the same ECU or on another ECU
- The RTE provides task scheduling, communication APIs, state management, exclusive areas, NvRam integration and a lot of other services

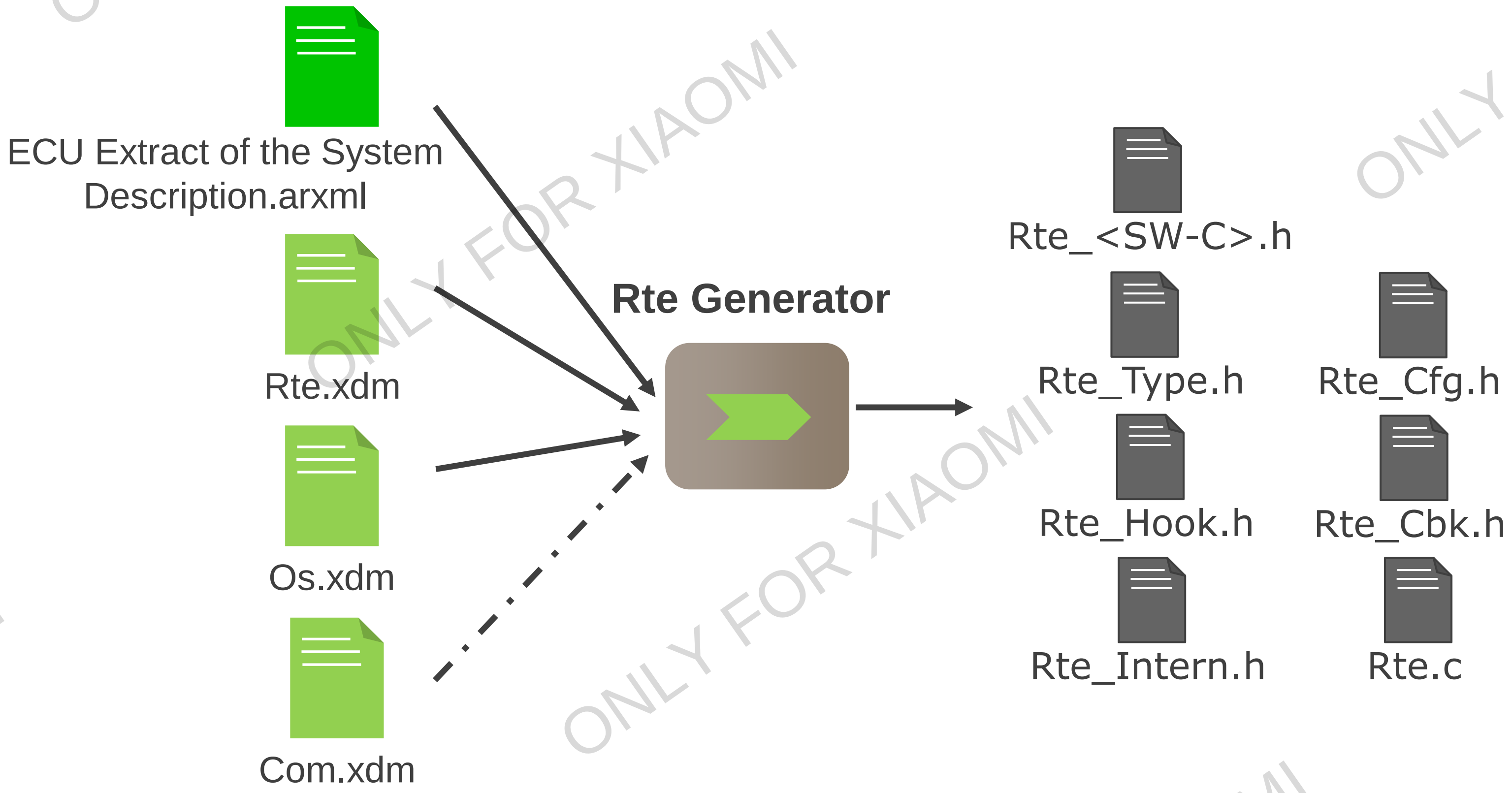


Workflow – Rte Generation



Workflow – RTE Generation

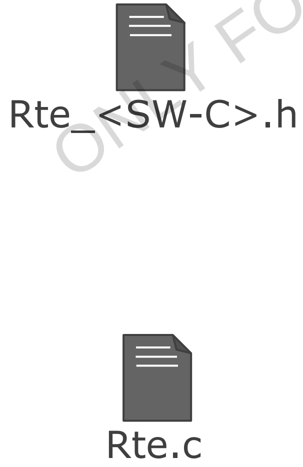
- ***RTE generation phase*** is used when the complete ECU Configuration Description is available
- Generated files: entire compilable RTE source code and additional information



Content of Output Files

- Application Header Files
  - API Mapping
  - Declarations of Runnable Entities
  - Declaration of API Functions
  - Declaration of Instance Handle

- Rte Source File
  - Implementation of API functions
  - Implementation of task bodies
  - Implementation of Rte_Start and Rte_Stop
  - Definition of buffers and queues
  - Definition of Component Data Structure
  - Implementation of COM callbacks



Elektrobit

Summary: Workflow for Rte Generation Process

Software Component Design

- Create Software Component Descriptions (*.arxml)
- Generate Header Files to check implementation of SWC‘s

System Design

- Map Software Components to ECUs
- Data Mappings

ECU Configuration

- Configure Os, Com (optional) and Rte

RTE Generation

- Use System Description, Os, Rte and Com Configuration to generate Rte

RTE events and event mapping

Elektrobit

Event overview

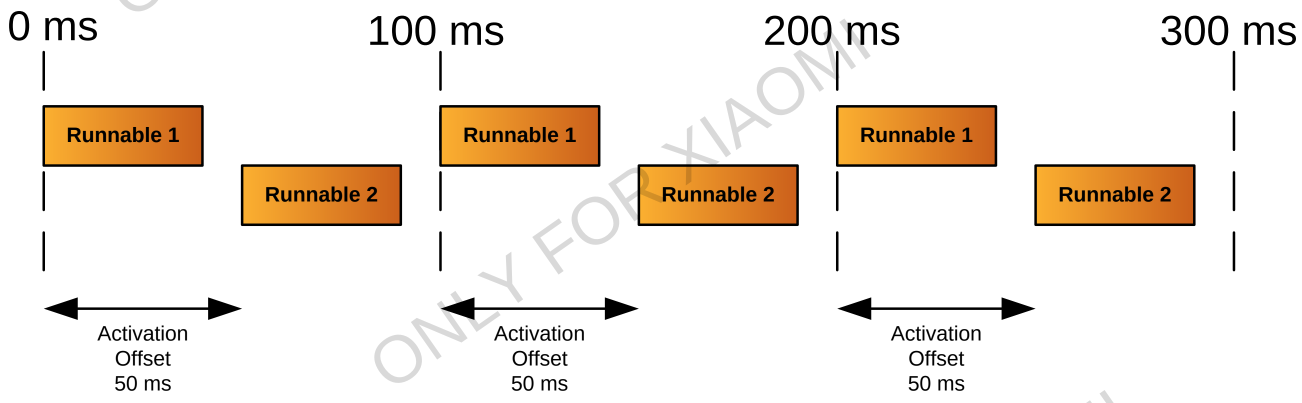
- Runnables will not be executed on their own – they must be triggered by an ***RTE Event***

- General events:
  - ***TimingEvent***
- Sender/Receiver events:
  - ***DataReceivedEvent***
  - ***DataReceiveErrorEvent***
  - ***DataSendCompletedEvent***
  - ***ModeSwitchedEvent***
- Client/Server events:
  - ***OperationInvokedEvent***
  - ***AsynchronousServerCallReturnsEvent***

– The mechanism to schedule BSW Mainfunctions is also part of the RTE. The corresponding event type is ***BswTimingEvent***

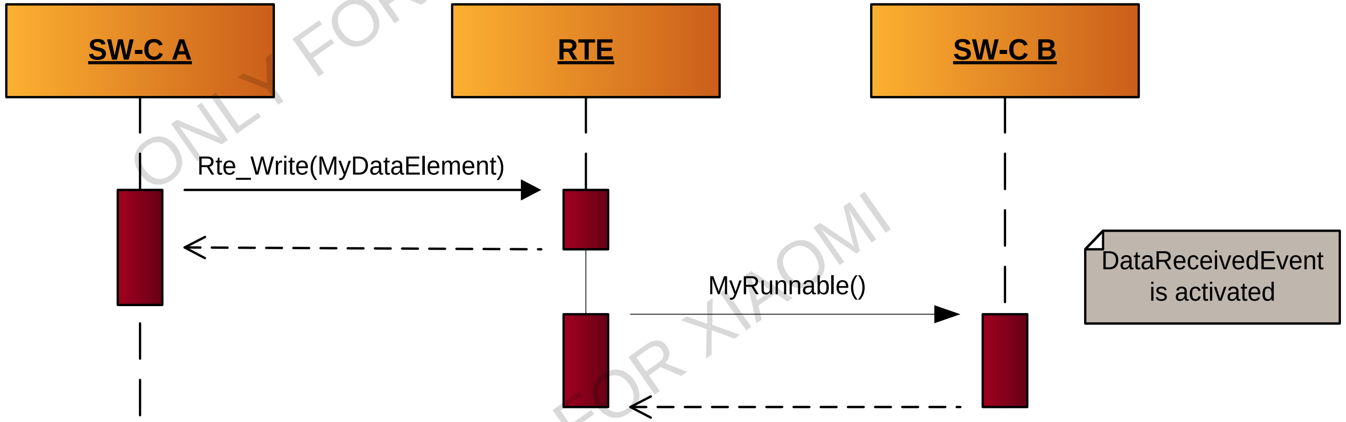
Timing event

- A ***TimingEvent*** is used to trigger a Runnable periodically
- An optional ***Activation Offset*** can be specified to avoid starting many runnables exactly at the same time (distributes CPU load)
- ***Example:*** Two TimingEvents have a period time of 100 ms. Second TimingEvent has an activation offset of 50 ms:



Data receive event (DRE)

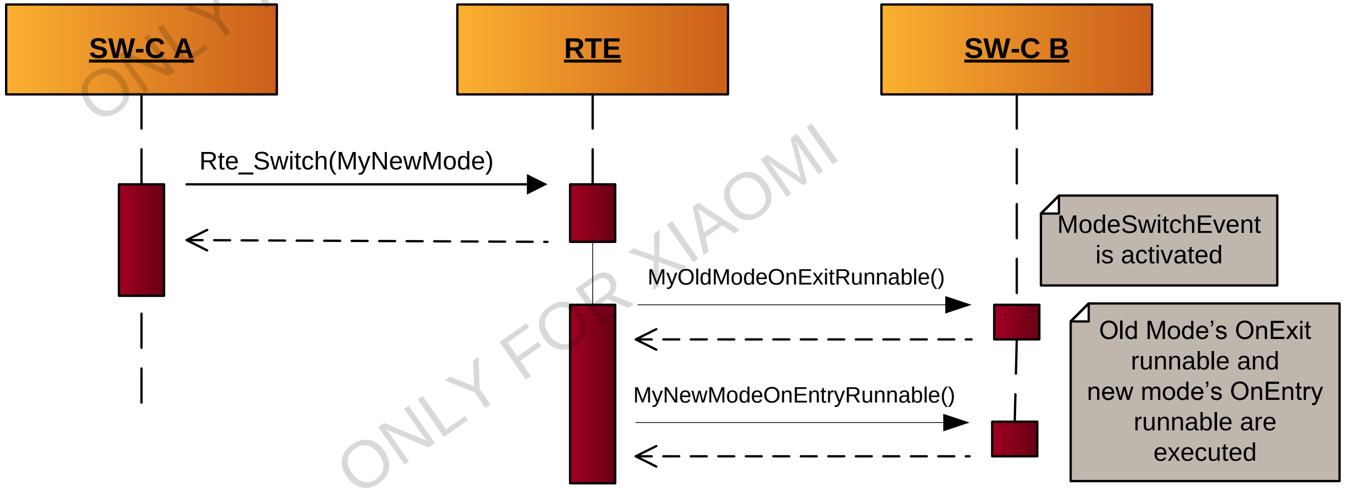
- A ***Data Received Event*** connects a certain DataElement in a port to a Runnable
- Upon data reception, the RTE will be trigger the Runnable as soon as possible
- If many DataReceivedEvents are connected to the same Runnable, it is possible to determine which one triggered the execution if the ***activating event feature*** is enabled
- If the DataElement comes from the network bus, an ***ComNotification*** RX indication must be configured in the Com module to notify the RTE



(*) *activation reason:* EB-specific, additional argument provided by RTE to the runnable triggered by that event (also for TimingEvent)

Mode switched events

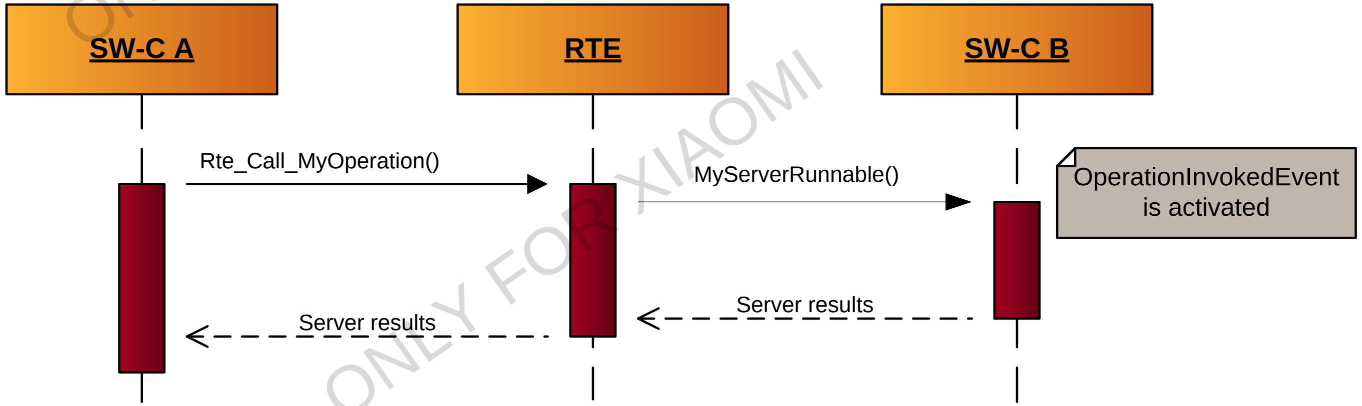
- A ***Mode Switched Event*** triggers a Runnable when a system state change has occurred
- State changes can only be broadcast inside one ECU
- A runnable can be triggered either ***on exit*** or ***on entry*** of a mode
- ***Mode switching*** will be explained in more detail later on…



Operation Invoked Event (OIE)

- An ***Operation Invoked Event*** triggers a server Runnable when a client makes a server call
- Client calls can be ***synchronous*** or ***asynchronous***
- Client/server will be explained in detail later on…

- Example of a synchronous server call:



Runnable Entity Mappings - Assigning events to Os tasks

- Multiple RteEvents can trigger the same runnable
  - ***Example:*** A TimingEvent and a DataReceivedEvent can trigger the same runnable

- The RteEvents (and implicitly the Runnables) must be ***mapped*** to Os tasks – so called ***Runnable Entity Mapping***

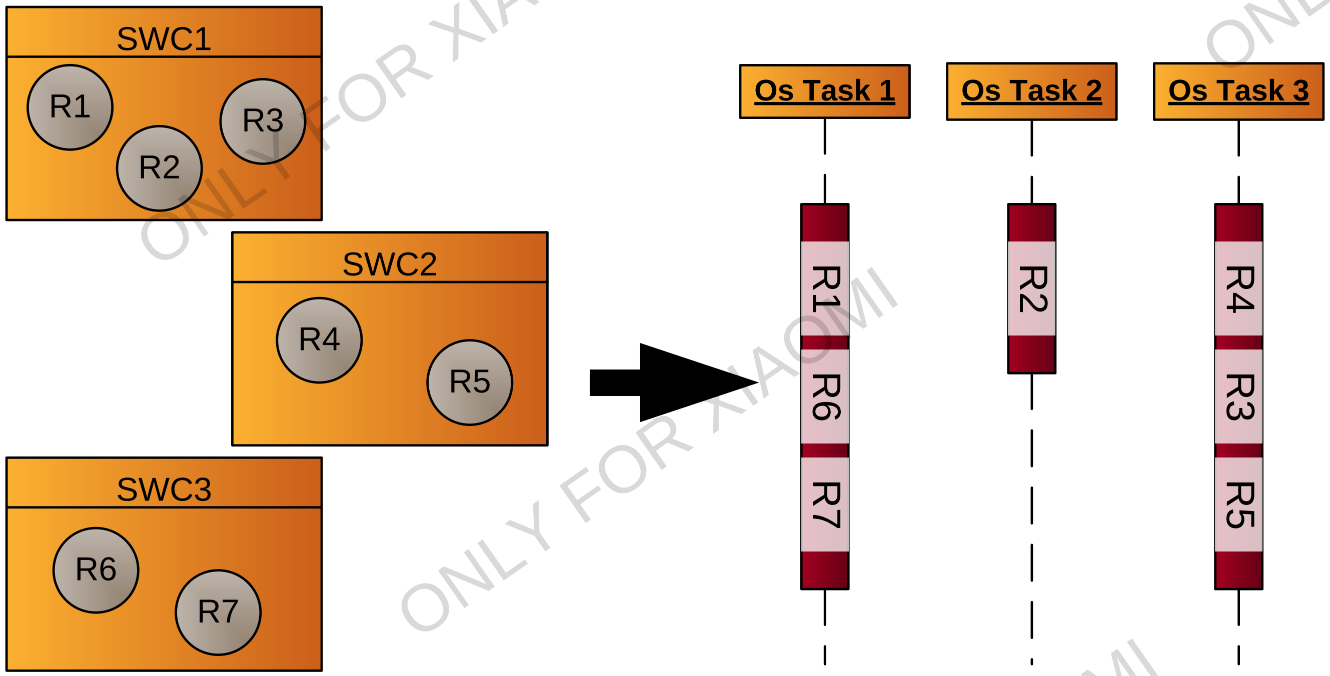
- Several RteEvents (Runnables) can be mapped to the *same* OsTask. This results in different more or less complex ***mapping scenarios***

- Use of blocking calls requires ***extended Os tasks***

- Not all RteEvents require task mapping, e.g. direct call server runnable entities → more about this later

Runnable Entity mapping - *Assigning runnables to Os tasks*

- The mapping is done in the ***RTE Configuration Editor***
- The ***execution order*** of RteEvents (runnables) within one Os Task can also be configured



RTE Integration with Os

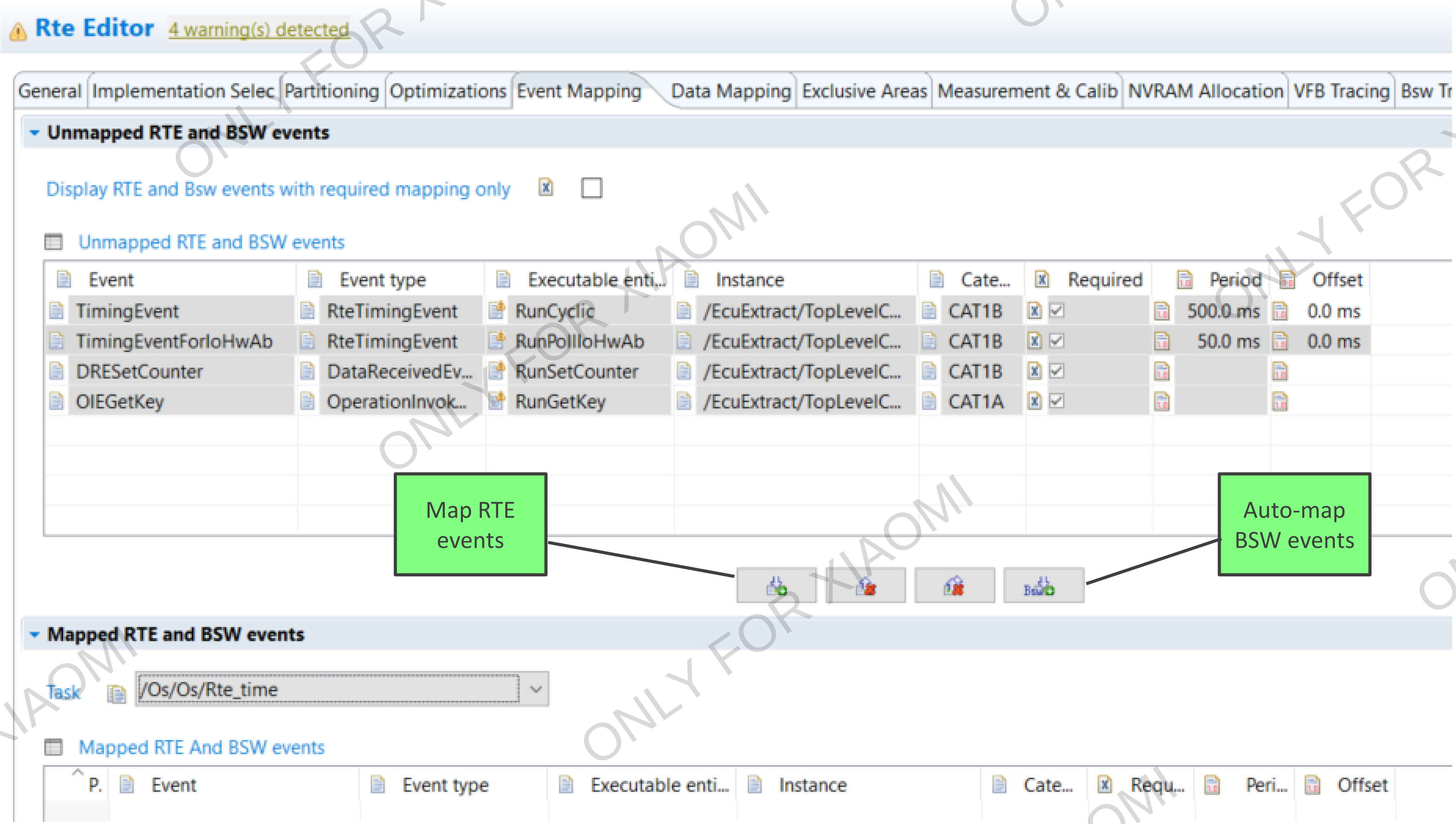
- The RTE will require some services from the Os:
  - ***Tasks*** – the different Runnables (functions) in the SW-Cs must be allocated to Os tasks
  - ***Counters, Events*** and ***Alarms*** – used to trigger the events in the system
  - ***Resources, Interrupt Locks*** and ***Spinlocks*** – used to implement ***Exclusive Areas***

- The counter, Os tasks and the allocation of events to the tasks must be configured prior running the RTE Generator

- Os configuration for the RTE Service needs:
  – When the RTE editor is closed – the RTE defines it’s “Service needs” in the System model
  – The unattended wizard “Service Needs Calculator” in EB tresos Studio will update the required Os configuration

Elektrobit

Runnable Entity mapping – example in EB tresos Studio



Advanced SWC concepts

Elektrobit

Section overview - Advanced concepts of SWCs

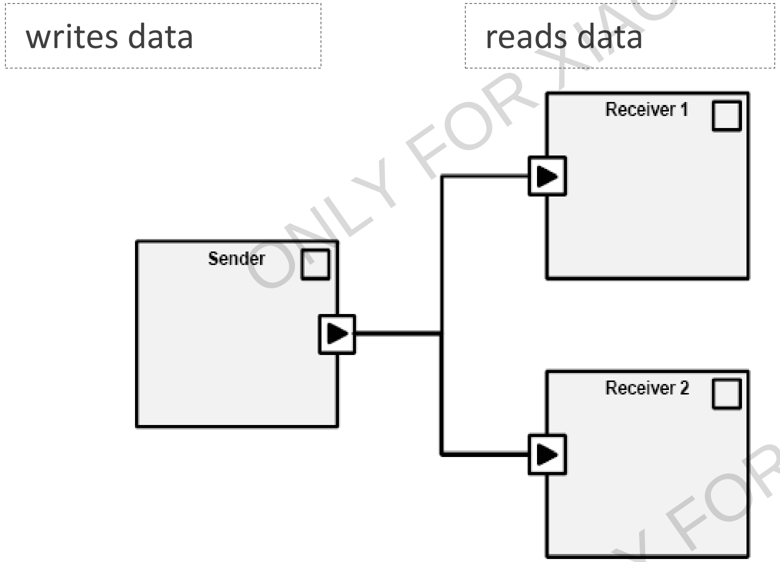
- Sender / Receiver
- Client / Server
- Interrunnable Variables
- Instantiation
- Exclusive areas
- Mode management
- Partioning

Advanced SWC concepts
Sender / Receiver Interfaces

Elektrobit

Sender/Receiver - *Quick review*

- We use Sender/Receiver interfaces for ***one way communication***
- Multiple ***DataElements*** (signals) can be bundled in one ***Interface***
- Senders use ***PPorts*** while the receivers use ***RPorts***
- Sender/Receiver communication can be ***queued*** or ***unqueued***



Explicit vs. Implicit communication

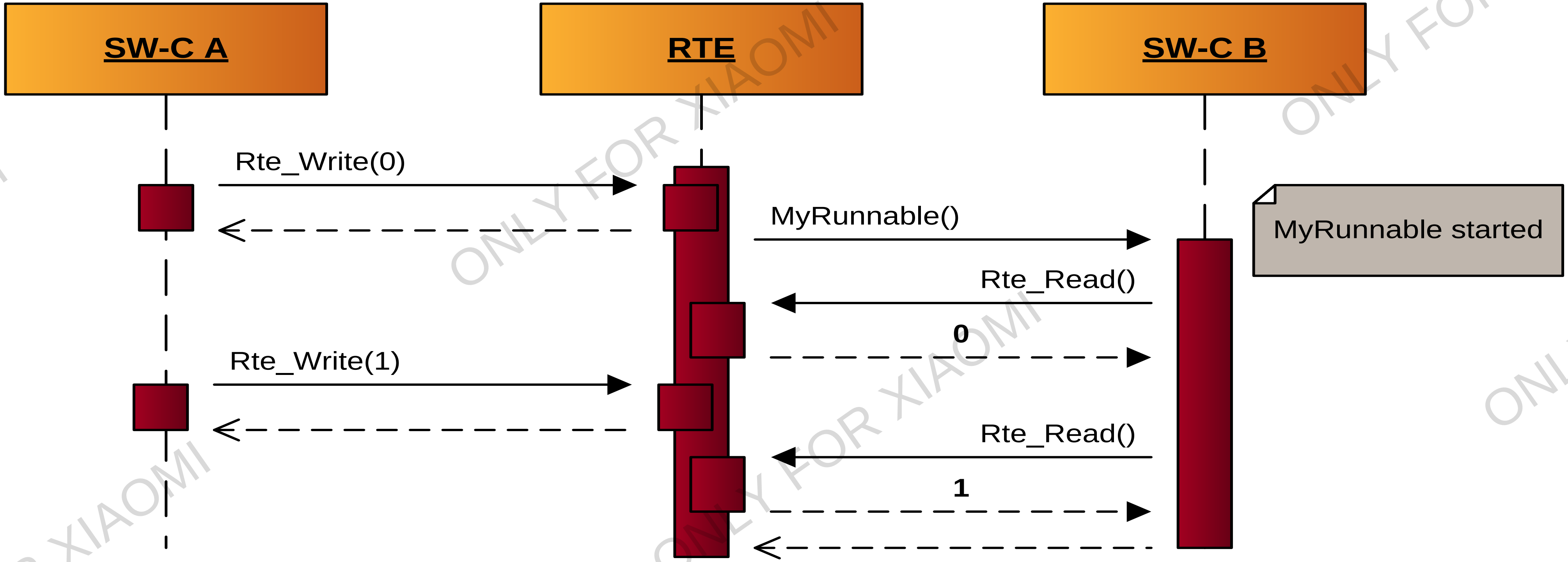
- The RTE provides two different semantics when using Sender/Receiver communication

  - ***Explicit Sender/Receiver*** (direct)
    - Sender: The RTE will always send the data directly at API invocation
    - Receiver: The RTE will always provide the latest data to the receiver

  - ***Implicit Sender/Receiver*** (buffered)
    - Sender: The RTE will buffer the sent data and send it when the sending Runnable returns
    - Receiver: The RTE will buffer the received data when Runnable starts

Explicit Sender/Receiver communication

- When using explicit sender/receiver communication, the receiving Runnable will always read the latest data provided by the sender



Explicit Sender/Receiver - Runnable APIs

```
runnable myRunnable {
   dataSendPoint
      MyPPort.MyDataElement
}
```

A ***DataSendPoint*** must be added to the Runnable to generate the ***Write/Send*** APIs

Explicit sending of unqueued data:

Rte_Write_MyPPort_MyDataElement(12345);

Explicit sending of queued data:

Rte_Send_MyPPort_MyDataElement(12345);

```
runnable myRunnable {
   dataReceivePoint
      MyRPort.MyDataElement
}
```

A ***DataReceivePoint*** must be added to the Runnable to generate the ***Read/Receive*** APIs

Explicit reception of unqueued data:

UInt16 myVar;

Rte_Read_MyRPort_MyDataElement(&myVar);

Explicit reception of queued data:

UInt16 myVar;

Rte_Receive_MyRPort_MyDataElement(&myVar);

Explicit Sender/Receiver - Runnable APIs

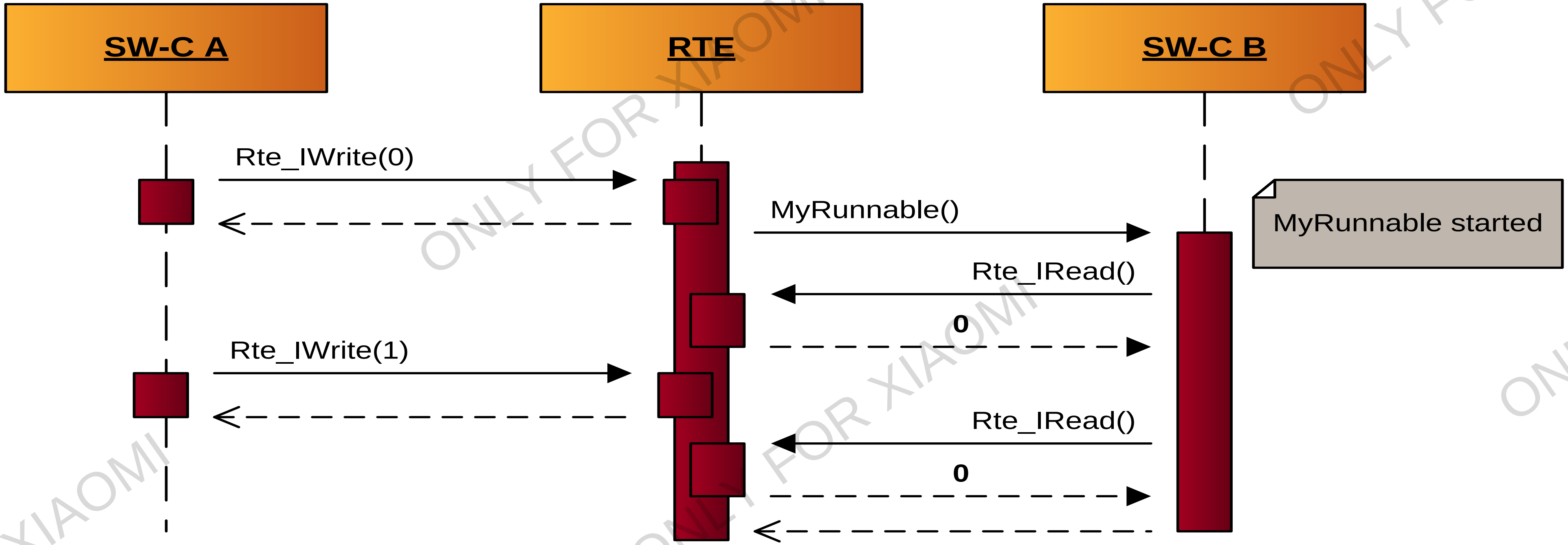
- When receiving queued data, you must check the return value:

```
UInt16 myVar;
Std_ReturnType retVal = RTE_E_OK;
while (retVal == RTE_E_OK)
{
  retVal = Rte_Receive_MyRPort_MyDataElement(&myVar);
  if (retVal == RTE_E_OK)
  {
    /* Queue element popped - do something with the data */
  }
  else if (retVal == RTE_E_NO_DATA)
  {
    /* Queue is empty (not an error) */
  }
  else if (retVal == RTE_E_LOST_DATA)
  {
    /* Queue overflow (error) */
  }
}
```

- (When receiving unqueued data ... well, just always check the return value.)

Implicit Sender/Receiver communication

- When using implicit sender/receiver communication, the receiving Runnable will always read the same data until the runnable returns
- Data will be buffered as a runnable entity-specific copy of the data element when the runnable started



IMPLICIT SENDER/RECEIVER - Runnable APIs

```
runnable myRunnable {
   dataWriteAccess
      MyPPort.MyDataElement
}
```

A ***DataWriteAccess*** must be added to the Runnable configuration to generate the ***IWrite*** API

Implicit sending of <u>unqueued</u> data:

```
Rte_IWrite_MyRunnable_MyPPort_MyDataElement(123);
```

NOTE: Data is sent when Runnable function exits

```
runnable myRunnable {
   dataReadAccess
      MyRPort.MyDataElement
}
```

A ***DataReadAccess*** must be added to the Runnable configuration to generate the ***IRead*** API

Implicit reception of <u>unqueued</u> data:

```
UInt16 myVar =
   Rte_IRead_MyRunnable_MyRPort_MyDataElement();
```

NOTE: Data was read when Runnable function started

It is not possible to write/read <u>queued</u> data in implicit mode!

Blocking vs. Non-blocking

- The Rte API functions for explicit **transmission** (Rte_Write()/Rte_Send()) are always non-blocking
  → They return immediately after transmission has been initiated

- Depending on the configuration, the Rte API functions for explicit **reception** (Rte_Read(), Rte_Receive()) can be
  - ***Blocking:*** API waits for new data to arrive or for a time-out to expire
  - ***Non-Blocking***: API returns immediately:
    - with the last received data (in case of unqueued data element prototypes), or
    - with the first value stored in the reception queue, or
    - with an error, indicating that no new data was available (in case of queued data element prototypes).

- For Sender/Receiver both Rte_Read and Rte_Receive APIs can be configured to become blocking

Elektrobit

Blocking Rte_Receive - Usage

- The code for a blocking Rte_Receive API will look like this:

```
UInt16 myVar;
Std_ReturnType retVal;
/* The following call will block until data arrival or
timeout: */
retVal = Rte_Receive_MyRPort_MyDataElement(&myVar);
if (retVal == RTE_E_OK)
{
  /* Queue element popped - do something with myVar*/
}
else if (retVal == RTE_E_TIMEOUT)
{
  /* API timeout (not an error) */
}
else if (retVal == RTE_E_LOST_DATA)
{
  /* Queue overflow (error) */
}
```

Blocking Rte_Receive - WaitPoints

- How to make Rte_Receive become blocking:
  - Configure a ***DataReceivedEvent*** for a queued ***DataElement*** in an ***RPort***
  - Add a ***WaitPoint*** referencing the DataReceivedEvent

Some important things to consider regarding WaitPoints:

- The WaitPoint should have a proper ***timeout value*** to avoid blocking forever
- Always check the status return value!
- Adding a WaitPoint will result in an ***extended Os task*** which is more resource consuming than a basic Os task!
- Mixing different kinds of RTE Events in tasks containing runnables with blocking APIs is not recommended. Could result in ***dead locks***!

- WaitPoints can also be used for other APIs – more about this later…

Sender / Receiver APIs - Summary

- A DataElement can be ***queued*** or ***unqueued*** depending on its ***SwImplPolicy*** attribute in the Software component description
- Sender/Receiver communication can be done with ***explicit*** (direct) or ***implicit*** (buffered) semantics
- To get the APIs from the RTE generator, each Runnable has to define ***Data Send/Receive Points*** (explicit) and ***Data Read/Write Accesses*** (implicit)
- Receiving APIs (both Rte_Read and Rte_Receive) can be set as ***non-blocking*** or ***blocking*** by defining a ***WaitPoint*** in combination with a ***DataReceivedEvent***.

Advanced SWC concepts
Client / Server Interfaces

Elektrobit

Client/Server Interfaces - *Quick review*

- Client/Server interfaces are used to model ***function calls*** from a client to a server
- A SW-C acting as a server uses ***PPorts*** while the clients use ***RPorts***
- Multiple ***Operations*** (functions) can be bundled in one ***Interfaces***
- Each operation may have 0 or more ***Arguments*** which can be of direction ***IN***, ***OUT***, or ***IN/OUT***
- Each operation may return 0 or more ***Application errors***

Synchronous vs. Asynchronous communication

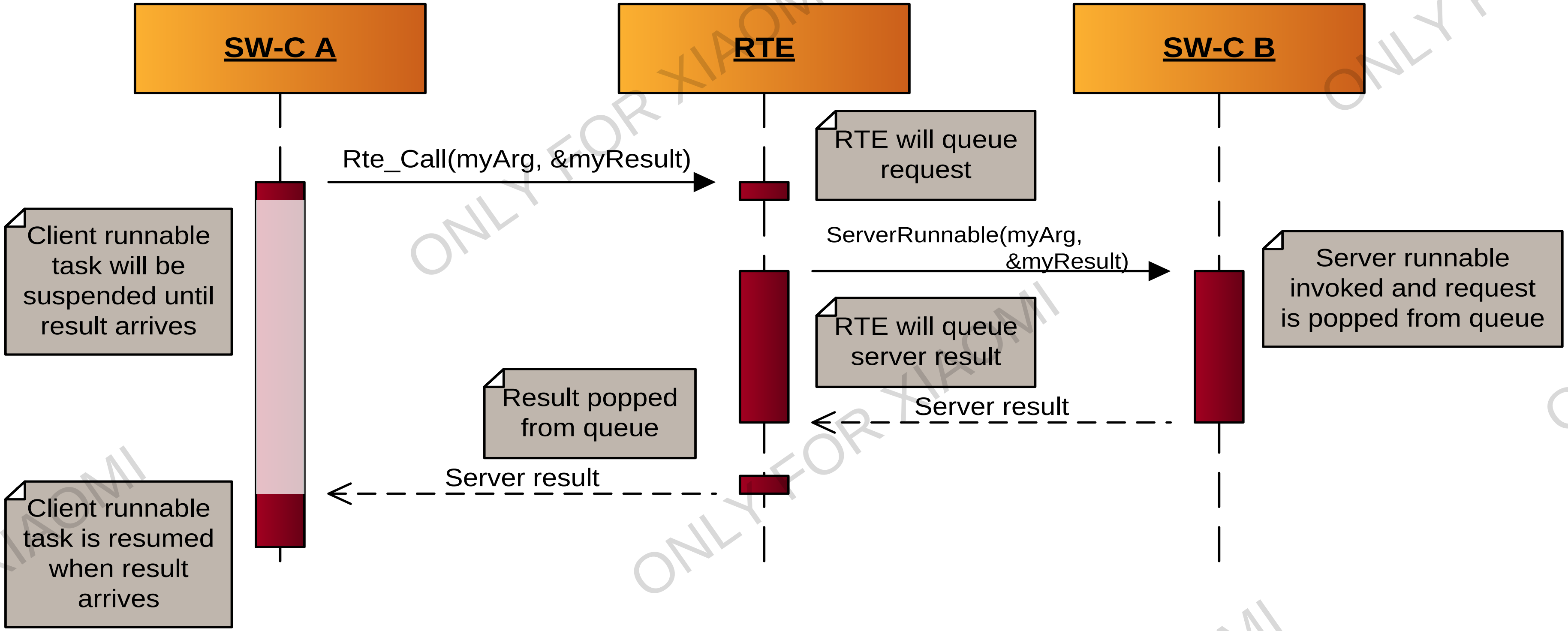
- The RTE provides two different semantics when using Client/Server communication:

  – ***Synchronous Client/Server***
    - The client software component calls an operation synchronously (Rte_Call()
    - The Rte API call generated for the call **blocks**
      – until the server execution has finished and the results are available, or
      – until an error occurs.will ***block*** until server is finished with the request
    - A special case of synchronous call is the direct call

  – ***Asynchronous Client/Server***
    - The client will do the invocation in two steps:
      – First sending the server call Rte_Call() non-blocking
      – Then using Rte_Result() to wait for the answer
    - The answer can be retrieved via ***polling*** or ***blocking*** call or even by ***triggering a runnable*** when the result arrives

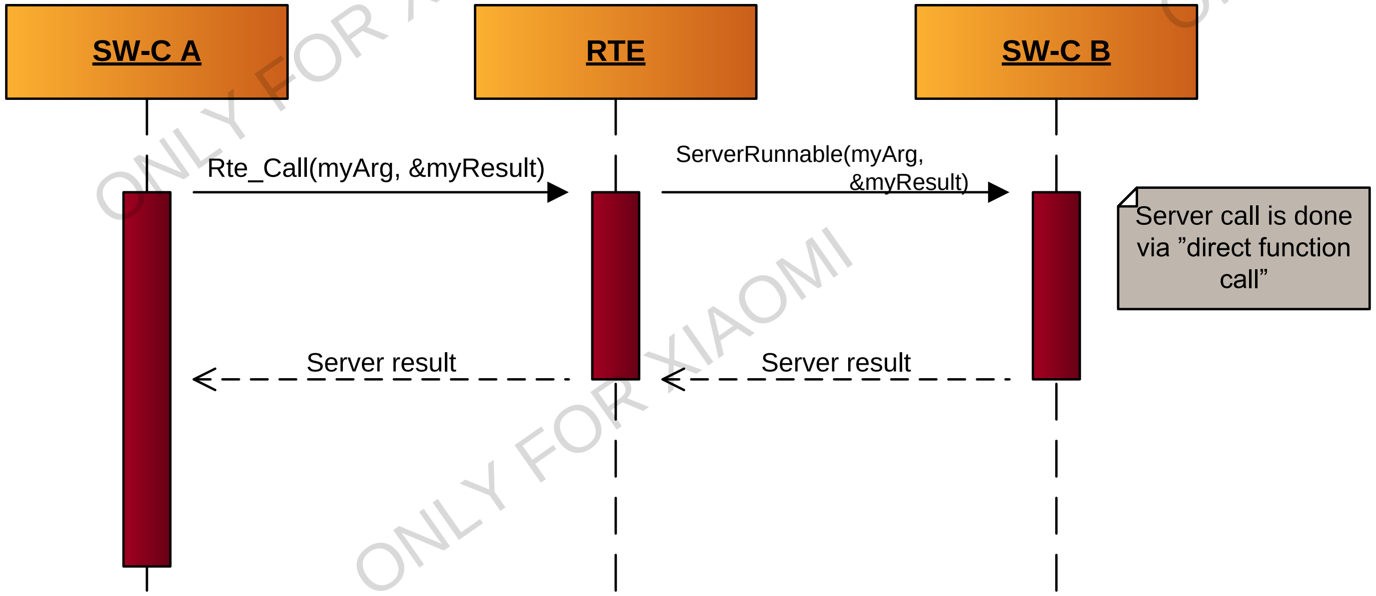
Synchronous Server Call - *Basic case*

- A ***synchronous server call*** will block until the server has finished processing the request
- Incoming requests are queued at server side
- ***Task context switch*** may take place



Synchronous Server Call - *Direct function call*

- ***Direct function call*** is the most efficient way of invoking a Client/Server runnable
- If server Runnable has attribute ***CanBeInvokedConcurrently*** set to “true”, it is considered as re-entrant
- The RTE will implement the server call as a ***direct function call*** when the OperationInvokedEvent of the ***RunnableEntity*** for the server ***is not mapped to a task***
  → The server function will be run in the client’s task context



Advanced SWC concepts
Interrunnable variables

Elektrobit

INTERRUNNABLE VARIABLES - Configuration

- ***Interrunnable variables*** are used to share data between runnables of the same SWC
- Interrunnable variables can be accessed in ***explicit*** (direct) or ***implicit*** (buffered) mode. Semantic is exactly the same as with sender/receiver explicit/implicit
- RTE will provide ***data integrity*** and ***synchronization*** functionality

```
internalBehavior mySwcIB for mySwcType {
  var myUInt8 explicit swcIRVar1 = 0
  var myUInt8 implicit swcIRVar2 = 42

  runnable myRunnable [0.1] {
    readVariables {
       swcIRVar1,
       swcIRVar2
    }
  }
}
```

INTERRUNNABLE VARIABLES - *APIs*

- The RTE will provide ***Rte_IrvRead()*** and ***Rte_IrvWrite()*** functions for explicit mode:

```
UInt16 myReadVar;
myReadVar = Rte_IrvRead_MyRunnable_MyIrv();
Rte_IrvWrite_MyRunnable_MyOtherIrv(12345);
```

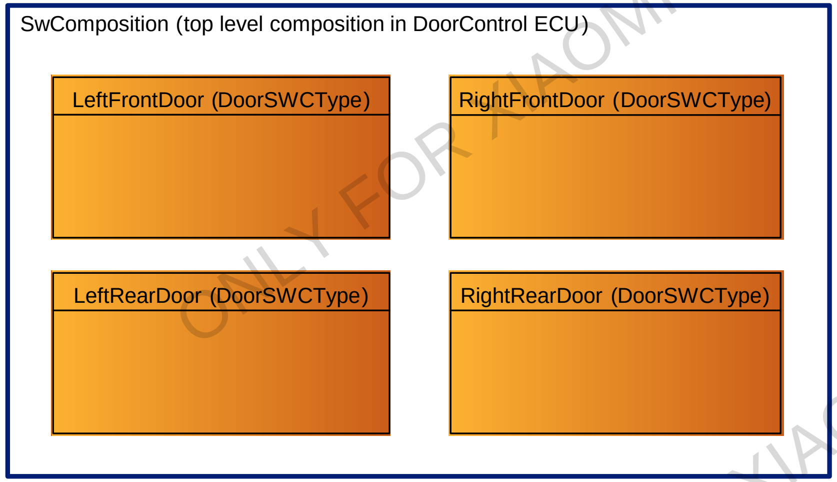
- The RTE will provide ***Rte_IrvIRead()*** and ***Rte_IrvIWrite()*** functions for implicit mode:

```
UInt16 myReadVar;
myReadVar = Rte_IrvIRead_MyRunnable_MyIrv();
Rte_IrvIWrite_MyRunnable_MyOtherIrv(12345);
```

Advanced SWC concepts
SW-C Instantiation

SW-C Instantiation – Quick review

- A component ***instance*** in a composition is called a ***ComponentPrototype***
- The ComponentPrototype references the SW-C Type
- If the SW-C Type supports ***multiple instantiation***, the same type can be instantiated in many ComponentPrototypes
- ***Example:*** Door lock application
  – the same SW-C can be re-used forAdvanced SWC concepts instanciation
  all four doors in a car by instantiating it four times in different Component- Prototypes

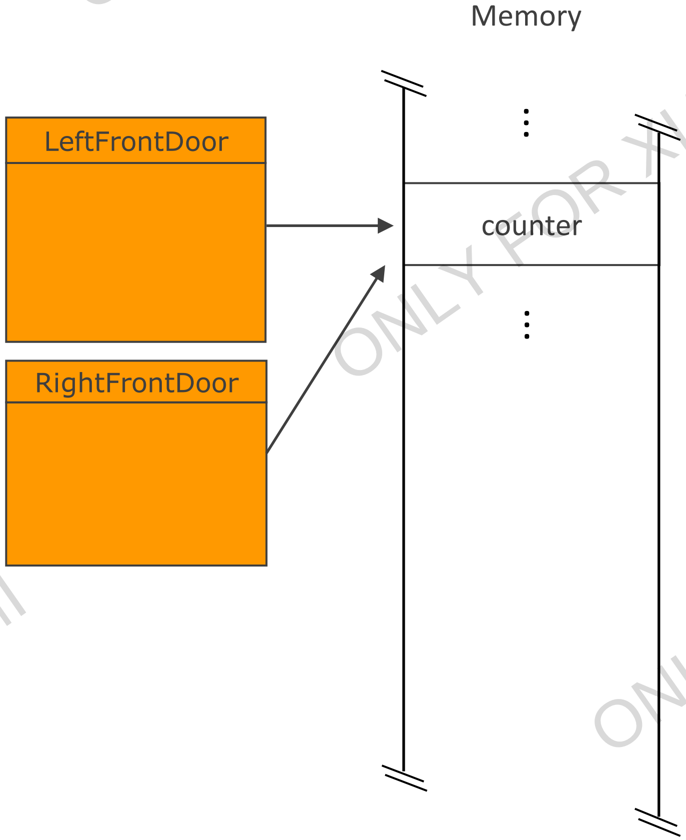


Elektrobit

Per Instance Memory

- Normal static variables are shared by all SWC instances!
- **Do not** use static variables together with multiple instantiation!

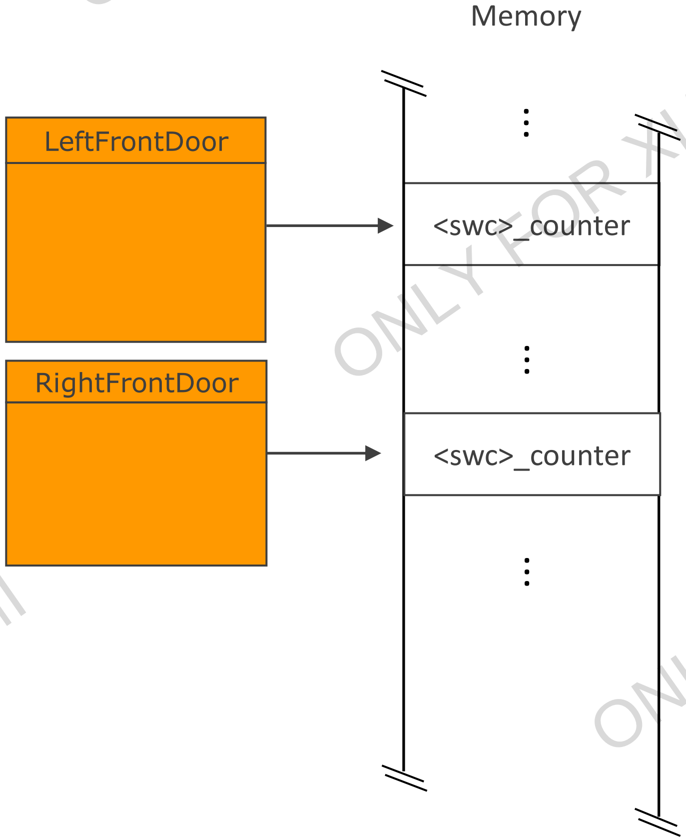
```
static uint8 counter;

void myRunnable(Rte_Inst_SWC
Rte_Instance) {
        /* … */
        counter++;
        /* … */
}
```



Per Instance Memory

- Use per instance memory for global variables
- The RTE allocates statically memory for each SWC

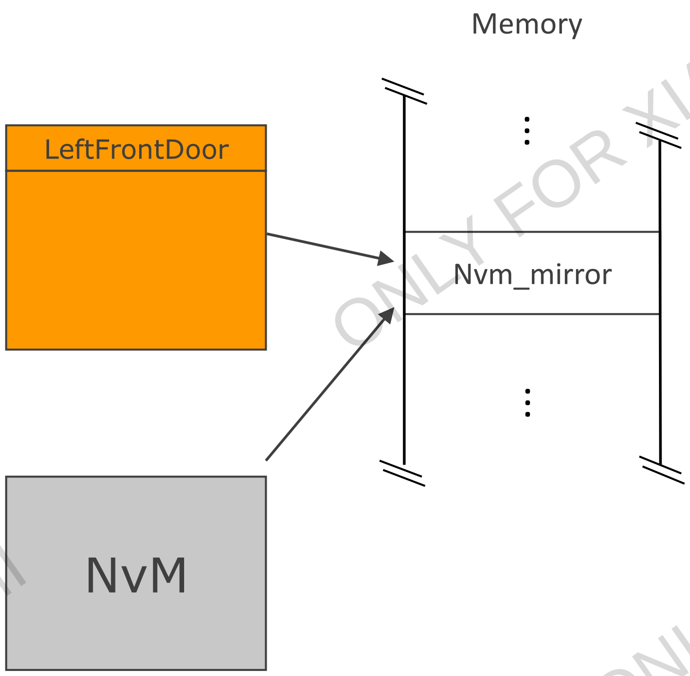
```
void myRunnable(Rte_Inst_SWC instance) {
        /* … */
        cpnt = Rte_Pim_Counter(instance);
        (*cpnt)++;
        /* … */
}
```



Per Instance Memory and NvM

- Per instance memory could also be used to access non-volatile memory
- Without multiple instantiation no instance handle is needed!

```
void myRunnable(void) {
        /* … */
        ptr = Rte_Pim_Data();
        /* … */
}
```



Advanced SWC concepts exclusive areas

EXCLUSIVE AREAS - *Protecting a code section*

- ***Exclusive Areas*** are used to protect sections of code (atomic operations) within one SW-C
- When entering and exiting an Exclusive Area, the ***Rte_Enter*** and ***Rte_Exit*** APIs are used:

```
/* Enter exclusive area */
Rte_Enter_MyExclusiveArea();
/* The protected code goes here... */
/* Exit exclusive area */
Rte_Exit_MyExclusiveArea();
```

- To enable the APIs for a runnable, add a ***CanEnterExclusiveArea*** reference to the Exclusive Area in your runnable configuration

```
exclusiveArea MyExArea
runnable MyRunnable uses MyExArea {
}
```

EXCLUSIVE AREAS - *Protecting an entire function*

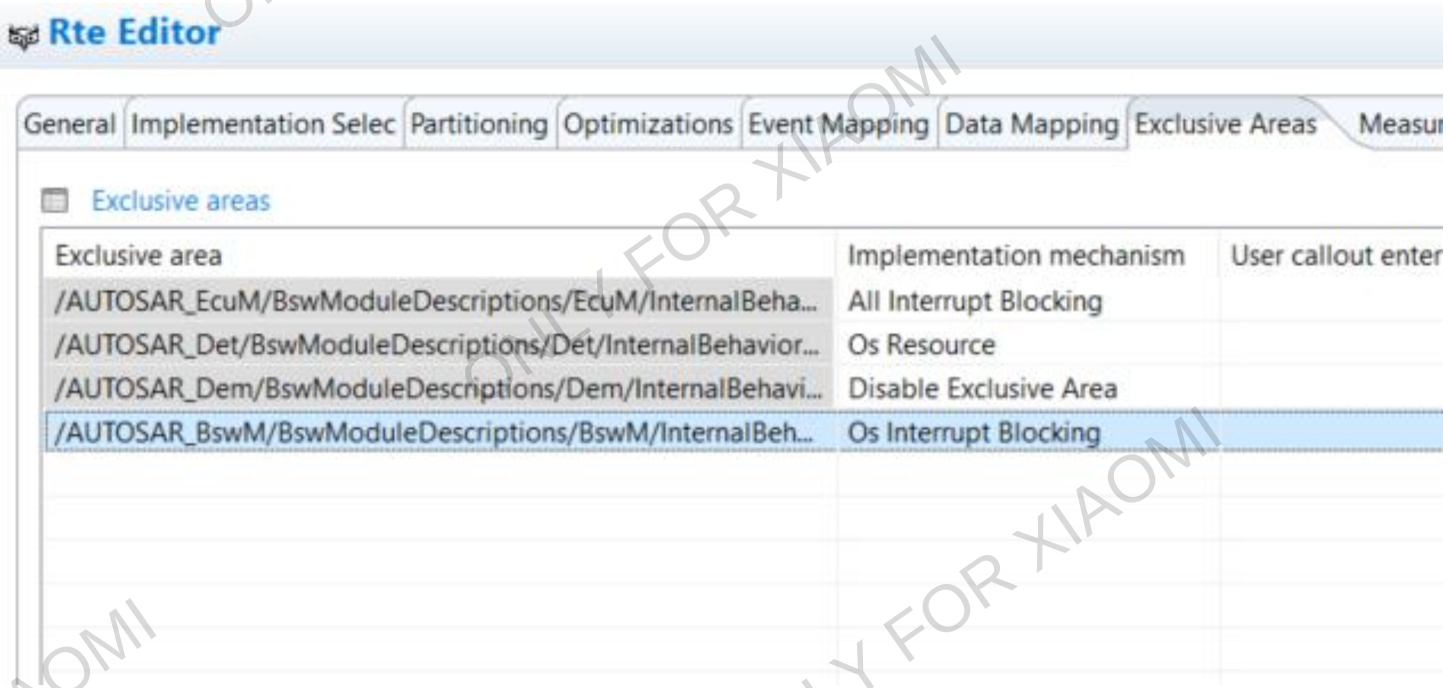
- An Exclusive Area can also be configured to protect an entire Runnable function
- To enable this feature, the attribute ***RunsInsideExclusiveArea*** is configured for the runnable:

```
internalBehavior MySwcIB for mySwcType {
  exclusiveArea MyExArea
  runnable MyRunnable in MyExArea {
    …
  }
}
```

- If using this approach, there is no need to use Rte_Enter/Exit

Exclusive Areas - implementation mechanism

- The actual implementation mechanism used by the Rte to protect the code section is configurable in the Rte configuration editor:



Advanced SWC concepts mode management

Elektrobit

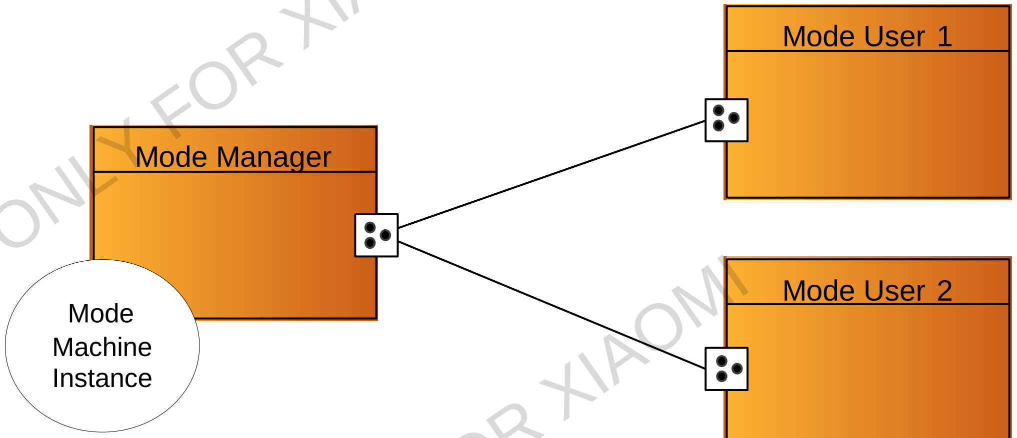
Mode Management

- ***Modes*** can be used to track system state changes
- Defining modes is a special case of sender/receiver communication
  - A ***ModeDeclarationGroup*** defines the different modes
  - In a Sender/Receiver interface, a ***ModeDeclarationGroupPrototype*** is added referencing the ModeDeclarationGroup

```
ModeDeclarationGroup "MyModes" {
          ModeDeclaration: STARTING
          ModeDeclaration: RUNNING
          ModeDeclaration: STOPPING
}
modeSwitch "MyModeInterface" {
          ModeDeclarationGroupPrototype "MyModePrototype" {
                    Type: /MyPackage/MyModes
          }
}
```

MODE MANAGEMENT - *Mode managers and users*

- One SW-C will act as a ***Mode Manager*** – the only one allowed to make state changes via its PPort
- The ***Mode Users*** will listen to mode changes by connecting their RPorts to the manager's PPort



- The RTE will provide a ***Mode Machine Instance*** to the manager which will take care of all mode changes

MODE MANAGEMENT - *Mode manager APIs*

- The Mode Manager will use the ***Rte_Switch*** API to initiate a mode change

```
/* Initiate mode change */
Rte_Switch_MyPPort_MyModePrototype(RTE_MODE_MyModes_STOPPING);
```

- To enable the API, a ***ModeSwitchPoint*** must be configured for the runnable

```
Runnable "MyRunnable" {
...
  ModeSwitchPoint "MyModeSwitchPoint" {
    Port: /MyPackage/MySwcType/MyPort
    ModeDeclarationGroupPrototype:
          /MyPackage/MyModeInterface/MyModePrototype
  }
}
```

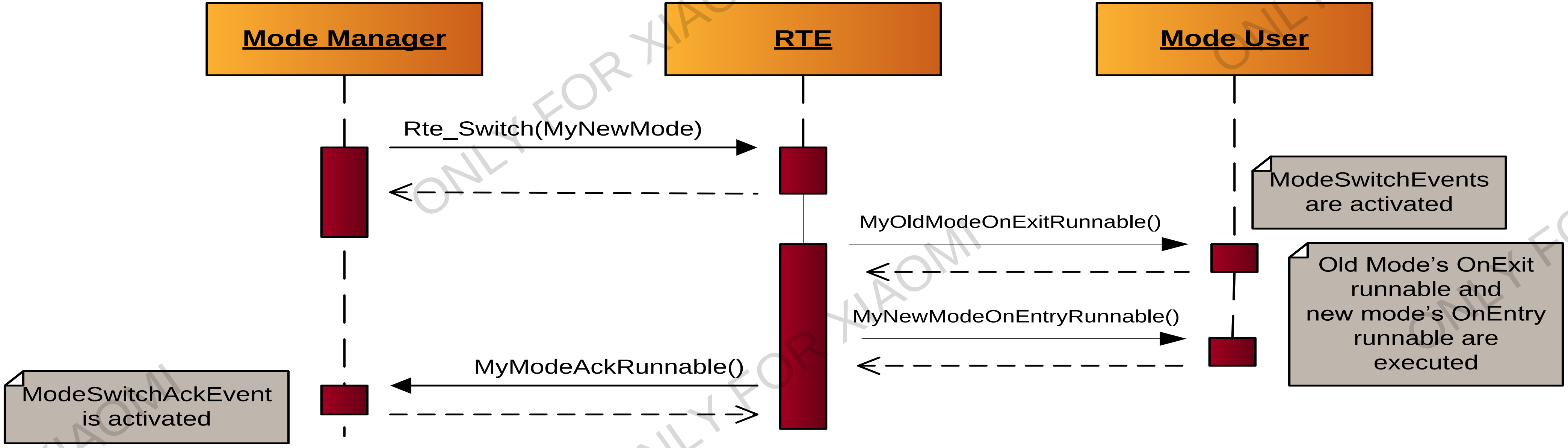
MODE MANAGEMENT - *Mode user APIs*

- The Mode Users will use the ***Rte_Mode*** API to poll current mode

```
Rte_ModeType_MyModes myModeVar;
/* Read current mode */
myModeVar = Rte_Mode_MyRPort_MyModePrototype();
```

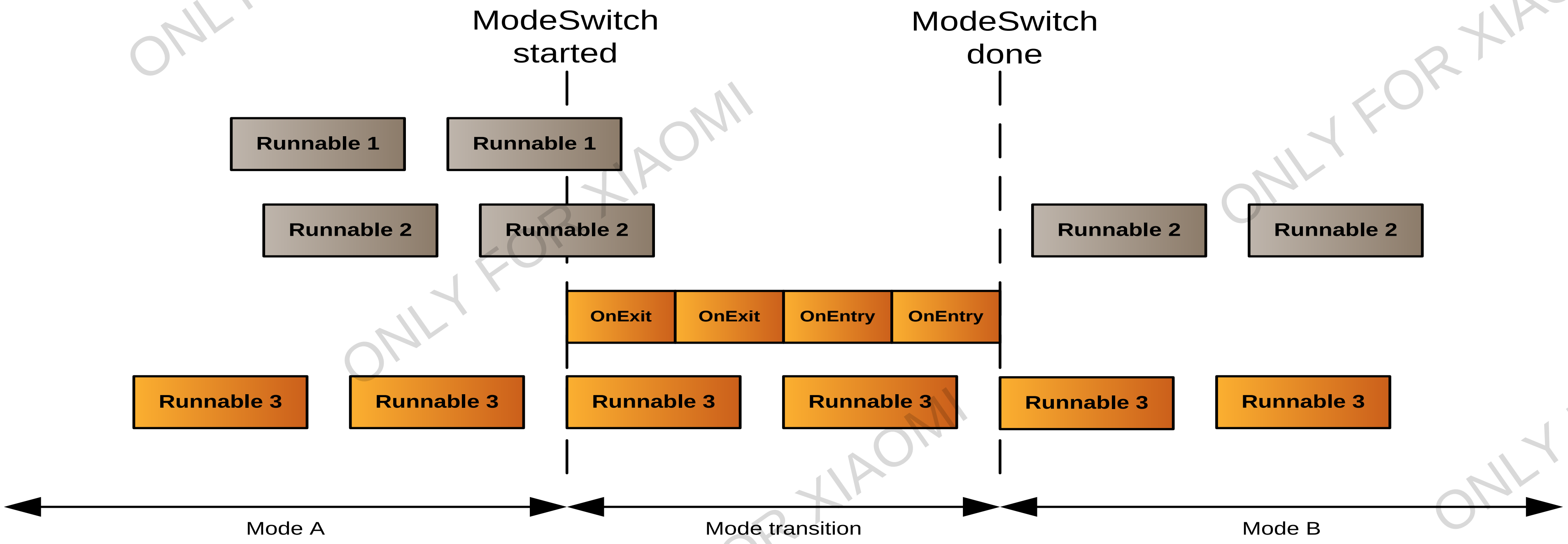
MODE MANAGEMENT - *OnEnter/Exit and acknowledgement*

- A mode user can use ***ModeSwitchEvents*** to trigger ***OnEnter*** and ***OnExit*** runnables upon mode change
- The Mode Manager can get an acknowledgement that all OnEnter/Exit runnables are finished via the ***ModeSwitchAckEvent***



MODE MANAGEMENT - Mode disabling dependencies

- ***Mode Disabling Dependencies*** are used to disable runnables during certain modes

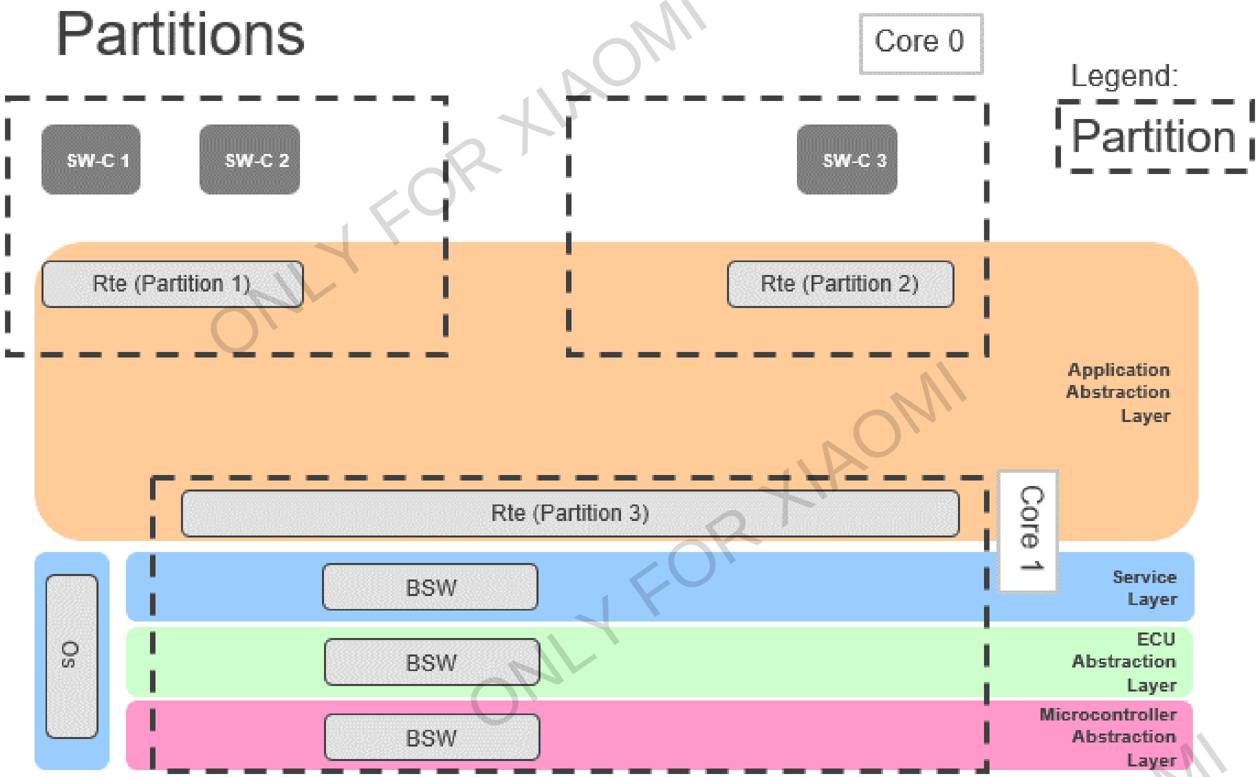


- During transition, Rte_Mode() returns ***RTE_TRANSITION_MyModeDeclarationGroup***

Advanced SWC concepts - Partitioning

Partitions

- The Rte supports mapping of software components to different partitions of an application.
- A partition itself is represented by an **Os application**. For each **Os application**, a separate **Rte** is generated.
- A partition must be allocated to exactly one core (but multiple partitions can be allocated to the same core)



Inter-partition communication

- The Rte offers the following possibilities to communicate between software components on different partitions:
  - The Inter Os Application Communicator (Ioc)
  - The Shared Memory Communicator (Smc)
  - Mix between the Ioc and Smc

- Independent of the chosen inter-partition communication mechanism the Rte offers the same API to the application. All solutions provide communication channels and channel groups (queued and non-queued) between partitions

Note: It is possible to combine intra- and inter-partition communication

Inter-partition communication with the Ioc

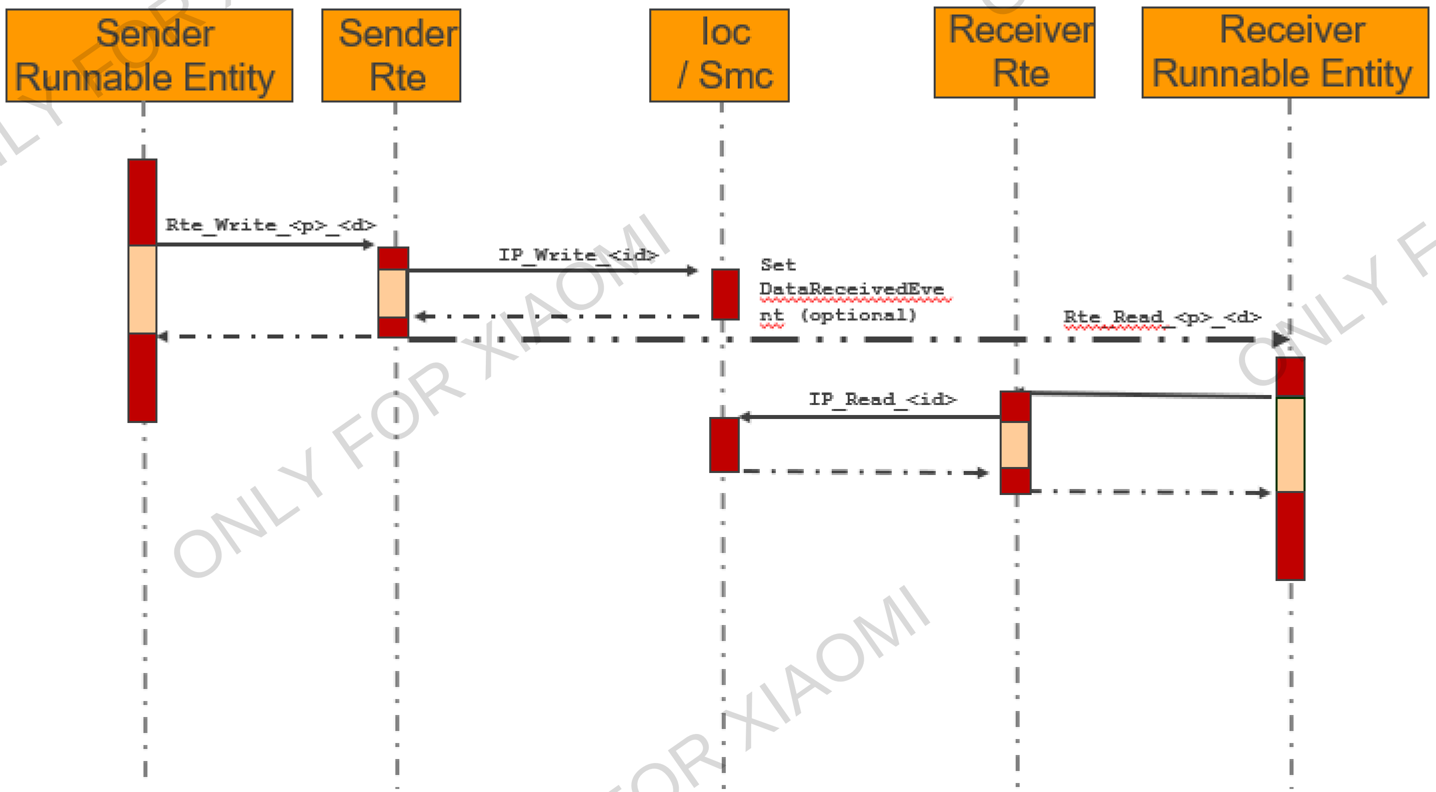
- The **Ioc** module is part of the Os.

- **Ioc** provides configurable channels and channel groups used to exchange data between Os applications.
- **Ioc** configuration is supported by the “Calculate Service Needs" wizard.
- The **Rte** associates the inter-partition channels and channel groups with the corresponding Ioc channels and channel groups.

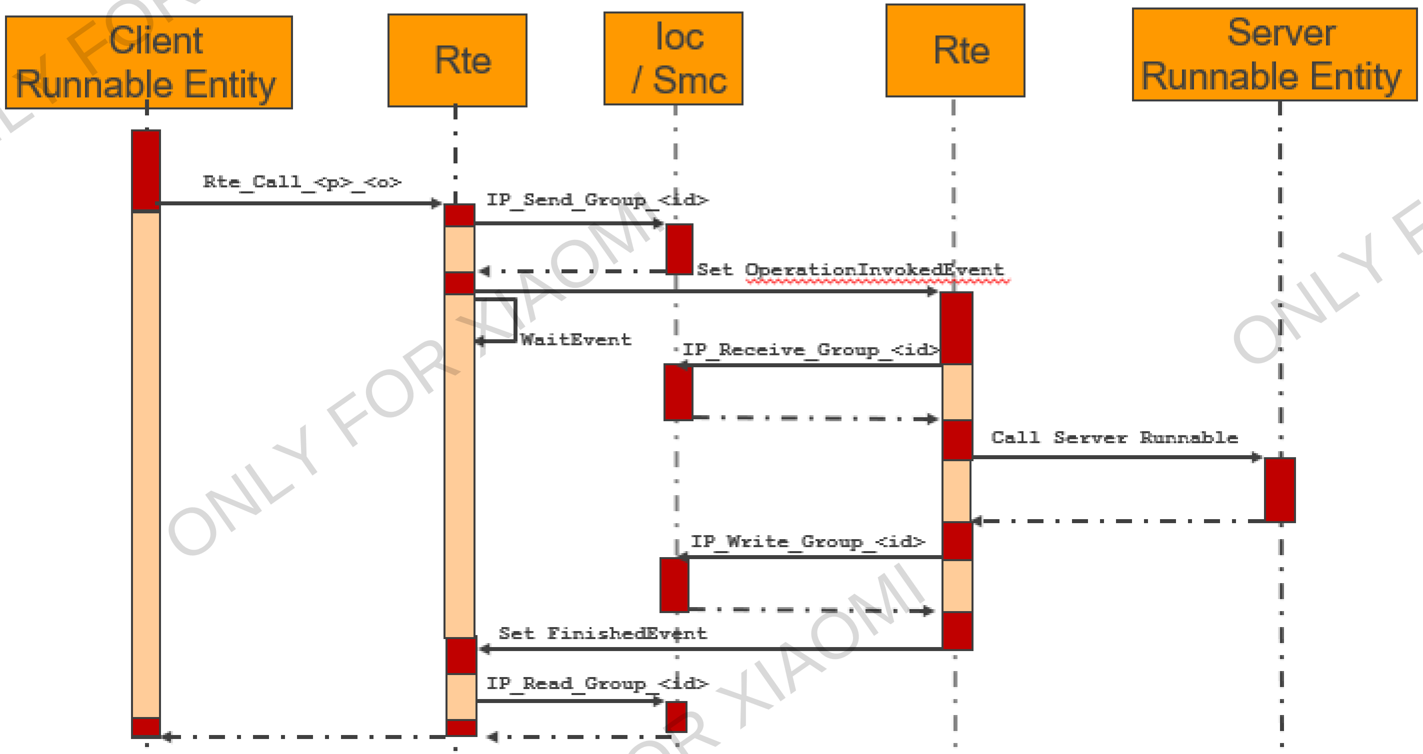
Inter-partition communication with the Smc

- The **Smc** is a mechanism provided by the Rte for inter-partition communication.
- **Rte** applies a shared memory concept. The Rte itself implements the functionality for the inter-partition channels and channel groups.
- When using the Smc the Rte generates additional files:
  - **Rte_Smc.h**
  - **Rte_Smc_Data_<partition>.c**

Intra-ECU/inter-partition sender/receiver communication



Intra-ECU/inter-partition client/server communication



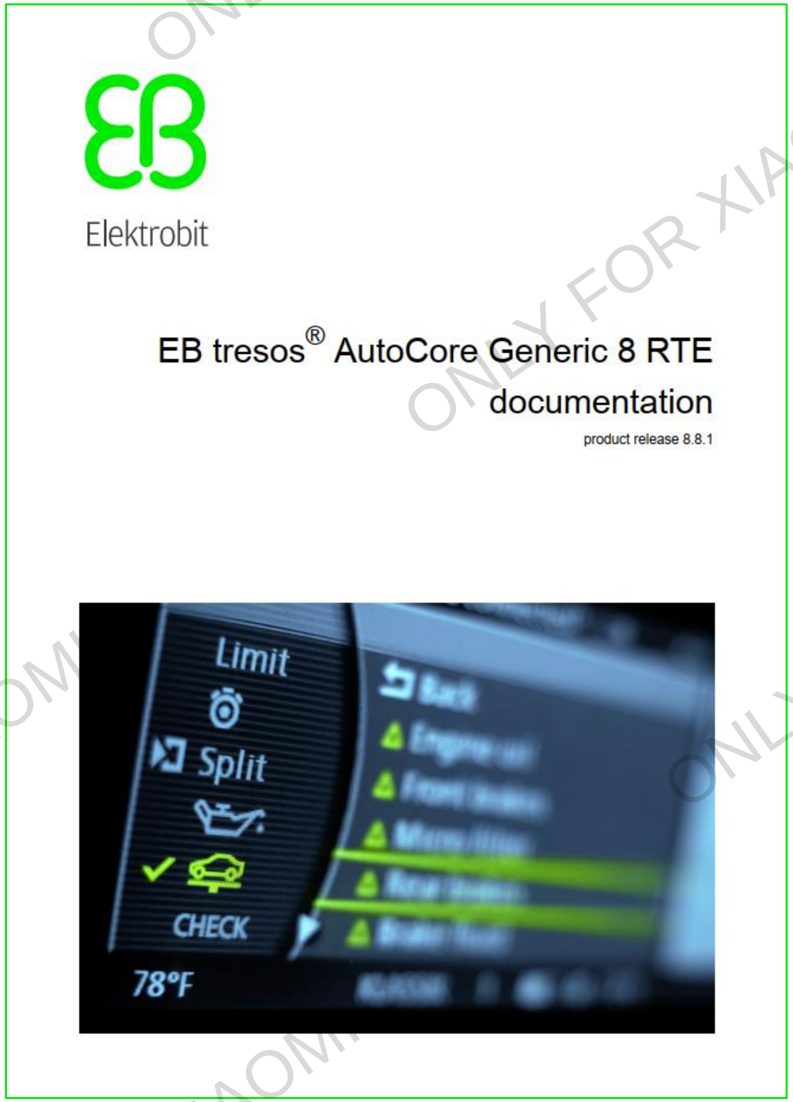
Hints for C/S inter-partition communication

- Consider the following when you use C/S communication across partitions:
- Inter-partition C/S communication:
  - has a significantly higher runtime than intra-partition direct call C/S communication
  - has influence on task mapping
    - task mapping is required for server runnable entities (in case “AllowInterPartitionDirectCall“ configuration parameter is missing or is false)
    - for synchronous server calls, client must be mapped to extended tasks
    - runnable entities with synchronous server call points might block other runnable entities mapped to the same task

- Think about timeout monitoring for C/S communication
- Think about using asynchronous C/S communication
- Avoid calling trees across partitions when possible

Limitations

- Limitations over multi partition/core exist for:
  - synchronous mode switching procedure
  - partial record support (signal degradation)
  - blocking Rte_Receive API
  - Com_InvalidateSignal API
  - data element invalidation
  - distribution of the Com onto multiple partitions
  - XfrmBufferLengthType via IOC
  - mode switched acknowledgment event
- More information can be found
  in AutoCore_Generic_RTE_documentation.pdf



Summary – Section Advanced concepts of SWCs

- Advanced concepts of SWCs
  - Sender / Receiver
  - Client / Server
  - Interrunnable Variables
  - Instantiation
  - Exclusive areas
  - Mode management
  - Partitioning

Get in touch!

sales@elektrobit.com
www.elektrobit.com

Elektrobit