

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 SW-C
- How to create a SW-C
- Connecting SW-Cs 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
- Partitioning

Advanced SW-C Concepts

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

Software Components

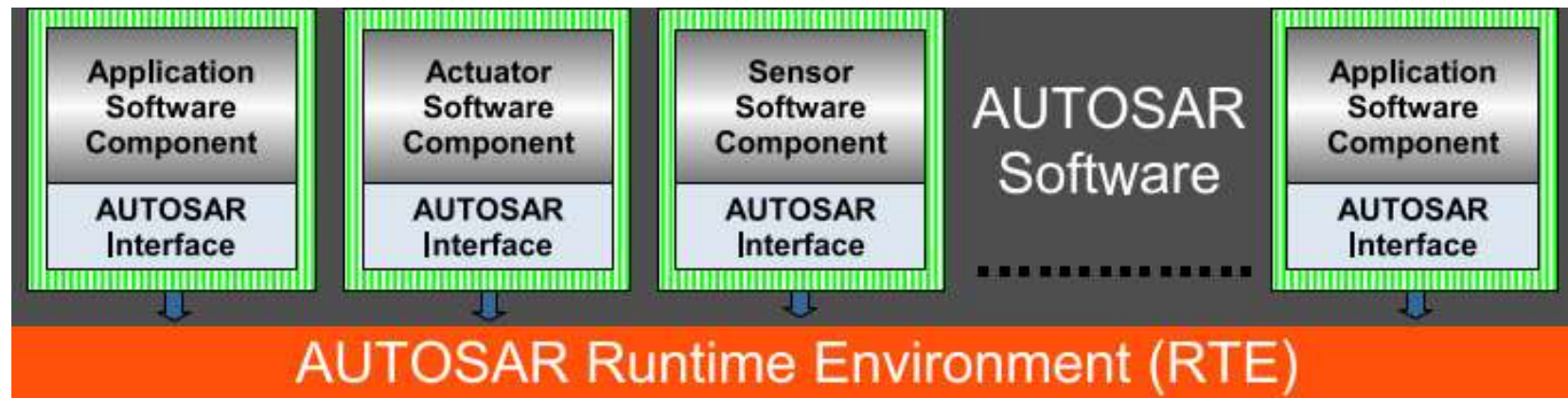


Elektrobit



The AUTOSAR Software Component (SW-C)

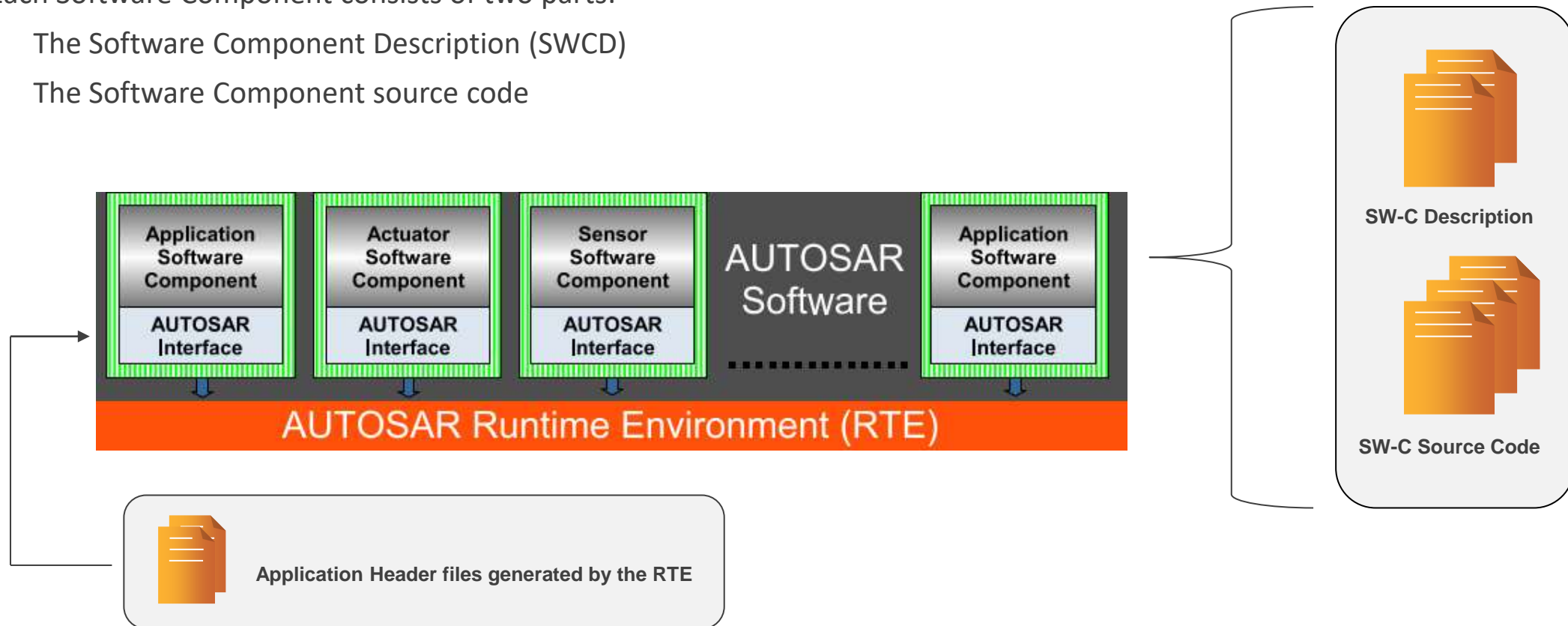
- Software components (SW-C) are used to structure the application part of the system functionality
- Undividable SW-C are called **Atomic Software Components**
- AUTOSAR distinguishes between Application SW-C, Sensor SW-C and Actuator SW-C



The AUTOSAR Software Component (SW-C)

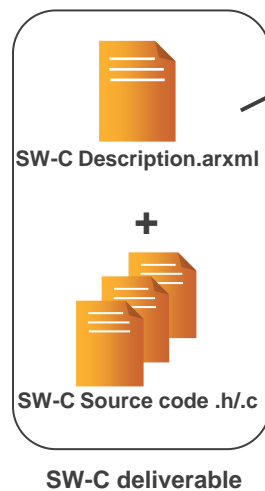
Each Software Component consists of two parts:

- The Software Component Description (SWCD)
- The Software Component source code



The Software Component Description (SWCD)

- The **SW-C Description (SWCD)** contains all description elements to define a Software Component (SW-C), e.g.
 - Used interfaces (Operations and data elements which are 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)



```
<?xml version="1.0" encoding="UTF-8"?>
<AUTOSAR xmlns="http://autosar.org/schema/r4.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://autosar.org/schema/r4.0 autosar_4-2-2.xsd">
  <AR-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>CounterDemo</SHORT-NAME>
      <ELEMENTS>
        <COMPOSITION-SW-COMPONENT-TYPE>
          <SHORT-NAME>TopLevelComposition</SHORT-NAME>
          <COMPONENTS>
            . . . .
```

Content of the SWCD

- AUTOSAR specifies the content and structure of the SWCD in the Software Component Template
- A **SW-C Description** (SWCD) consists of three main parts:
 - **SW-C Type Definition**
 - Defines all communication interfaces(APIs) that the SW-C will need from the RTE
 - **SW-C Internal Behavior**
 - Defines runnables (functions), variables, events and other internal data used by the SW-C
 - **SW-C 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, in the SWCD references to definitions are used which are defined on a project scope

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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation
2016-11-30	4.3.0	AUTOSAR Release Management	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• Minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation

A 1000+ page document

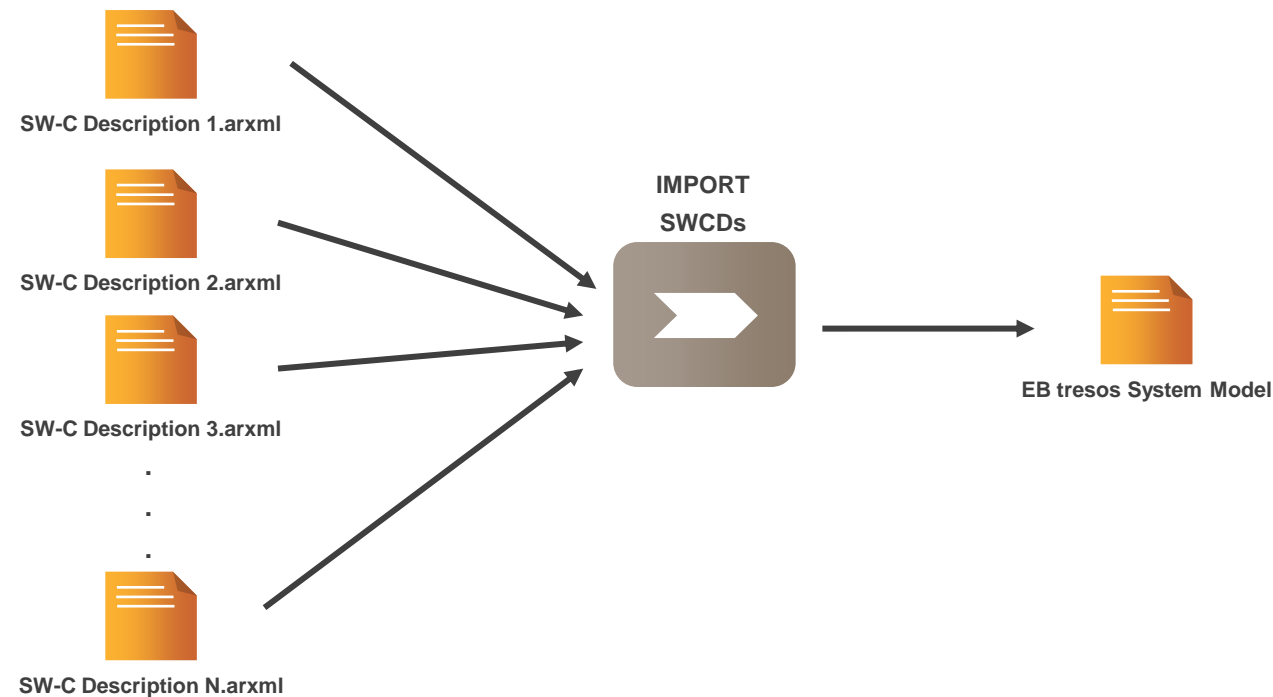


1 of 1069

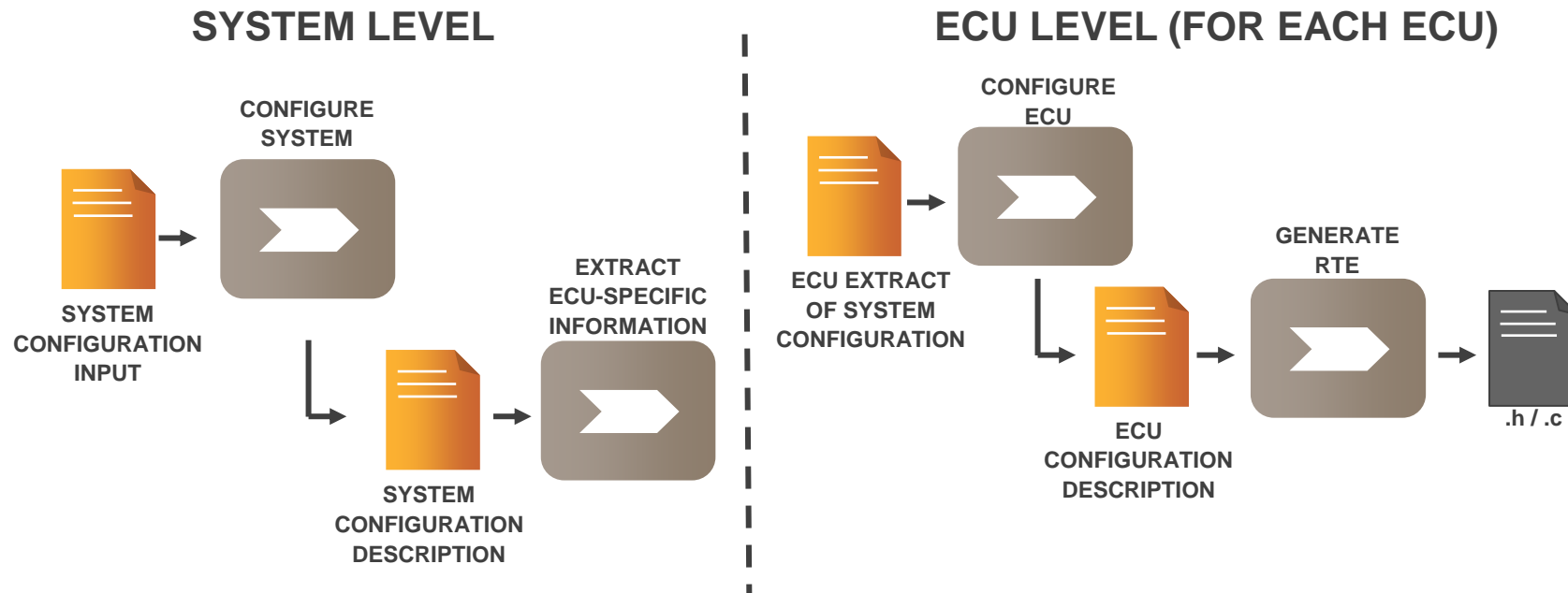
Document ID 062: AUTOSAR_TPS_SoftwareComponentTemplate
— AUTOSAR CONFIDENTIAL —

SWCDs are imported as part of the System Description ...

- 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!*

Basic Elements of an SW-C

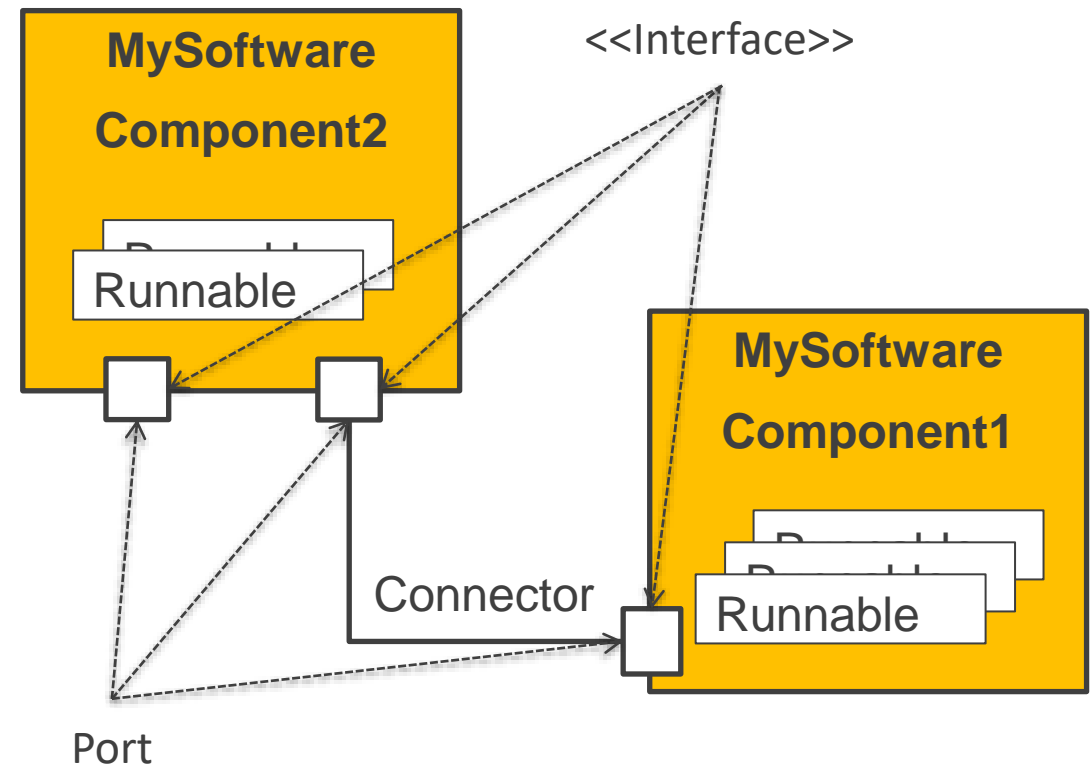


Elektrobit



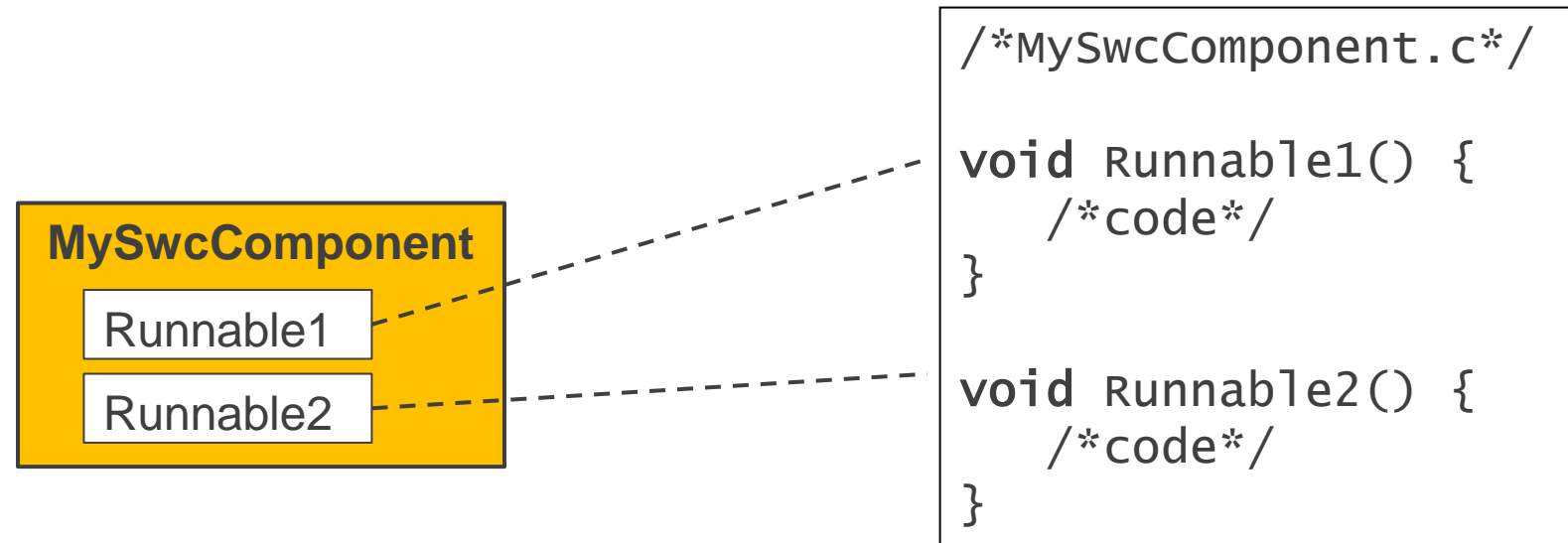
Basic Elements of an SW-C - Overview

- **Runnable:** (e.g. C) function to implement the functionality of the Software Component
- **Port:** point to connect a SW-C with other SW-C or services provided by/via the RTE
- **Interface:** specification of the communication via a Port
- **Connector:** connect SW-C Ports with other SW-C Ports or service Ports from the Basic Software (BSW)



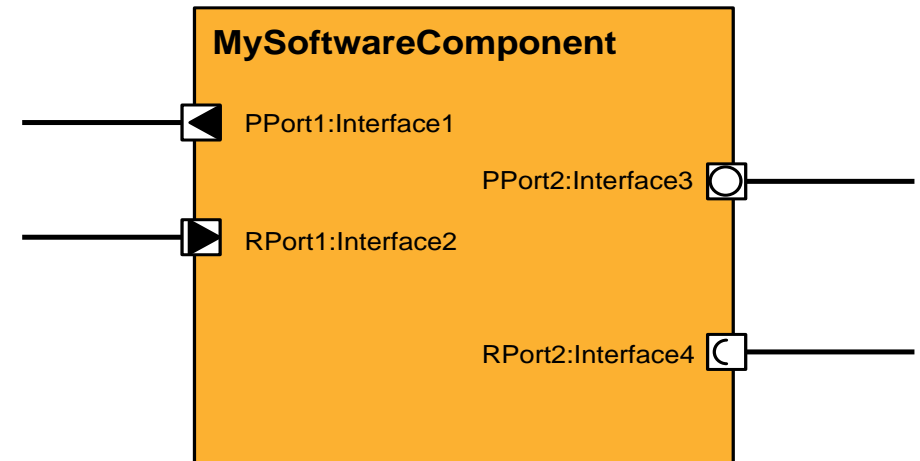
Basic Elements of an SW-C - Runnables

- 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.



Basic Elements of an SW-C - Ports

- **Ports** are used by an SW-C to communicate with other SW-Cs.
- We distinguish between the following port types:
 - **Provide port (P-Port):** The SW-C provides something via this port, e.g. data or a service.
 - **Require port (R-Port):** The SW-C requires something via this port.
 - **Provide-require port (PR-Port):** Combines the ability to provide and require services or data in one port.
- Each port implements one Interface
 - AUTOSAR language: “The port is typed by the interface”



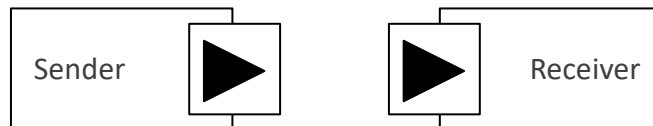
Basic Elements of an SW-C - Interfaces

Interfaces define

- the semantics of the communication and
- the kind of data that is transferred between the SW-Cs.

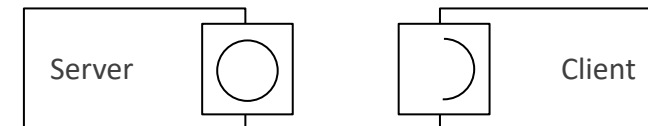
Semantics 1: *Sender/Receiver interface (S/R)*

- For broadcasting of signals (“DataElements”)
- One-way communication
- Many DataElements may be bundled in one S/R interface (though this is not recommended)





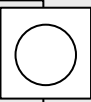
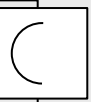



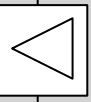


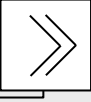



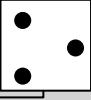
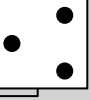
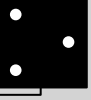
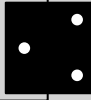






Semantics 2: *Client/Server interface (C/S)*

- For function invocations (“Operations”) 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 (though this is not recommended)

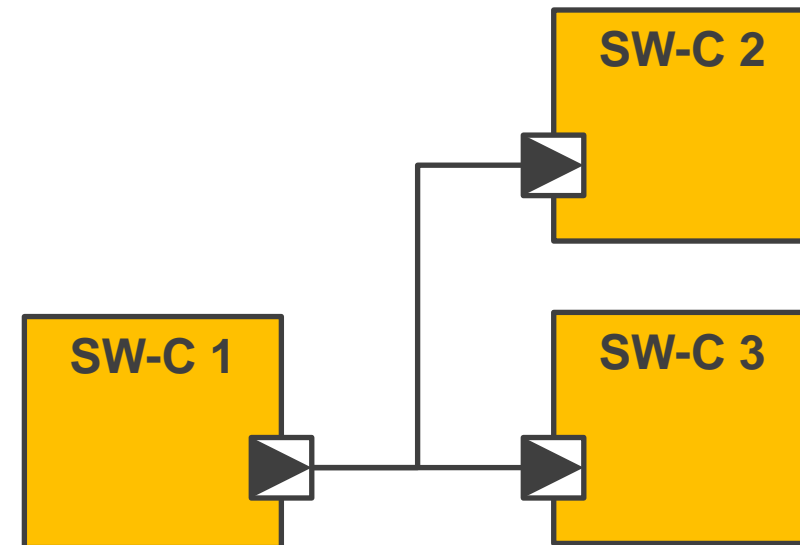


Port icons for different Interface types

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				

Basic Elements of an SW-C - Connectors

- **Connectors** connect (SW-C and BSW service) Ports.
- Ports (more precisely their Interfaces) must be compatible to be connected. More on that later ...
- Depending on the Interface one Port can have several Connectors (modelling 1:1, 1:n, n:1, and n:m communication).
- Connecting Ports is optional:
 - The RTE generates interface functions for unconnected Ports.
 - However, such interface functions do not initiate any communication to other SW-Cs or BSW modules.
 - PR-Ports are always considered as connected.



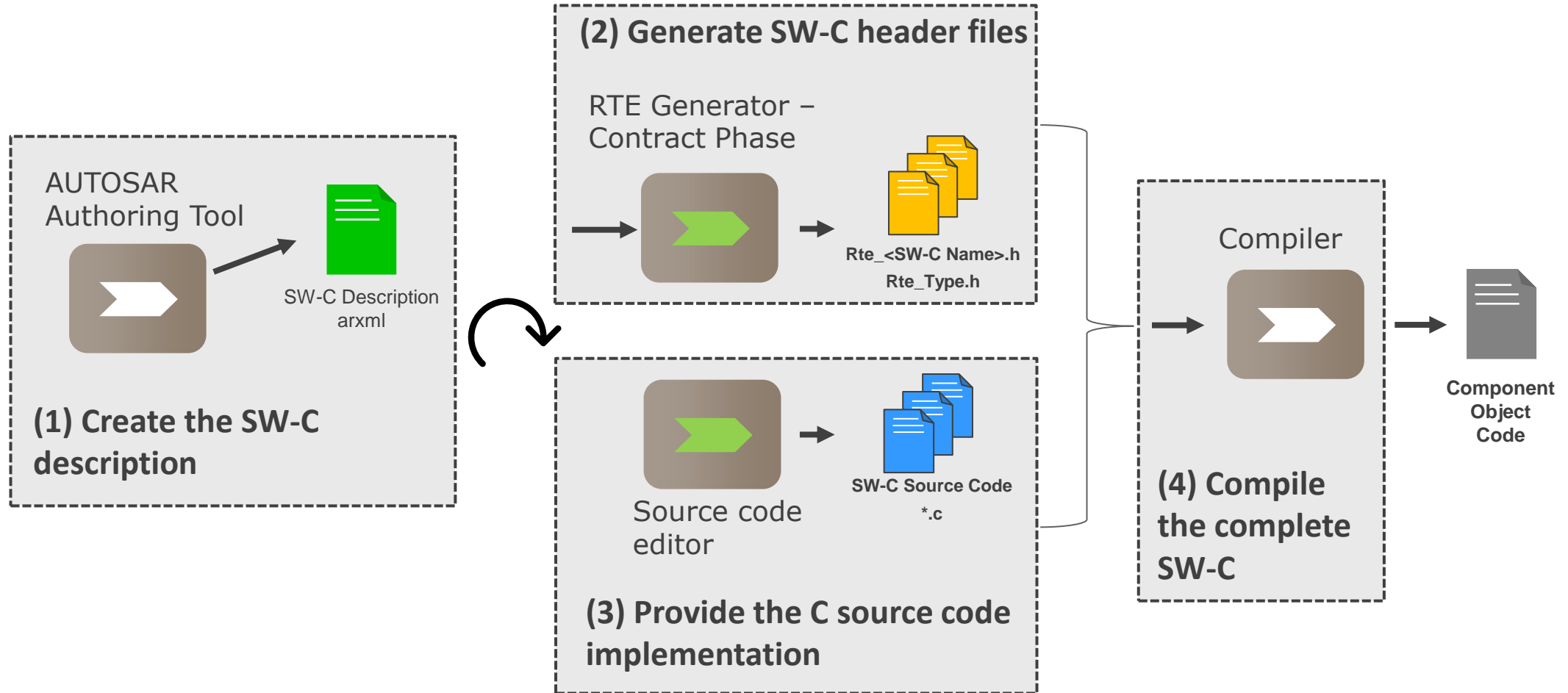
How to create an SW-C



Elektrobit



How to create an SW-C – Overview

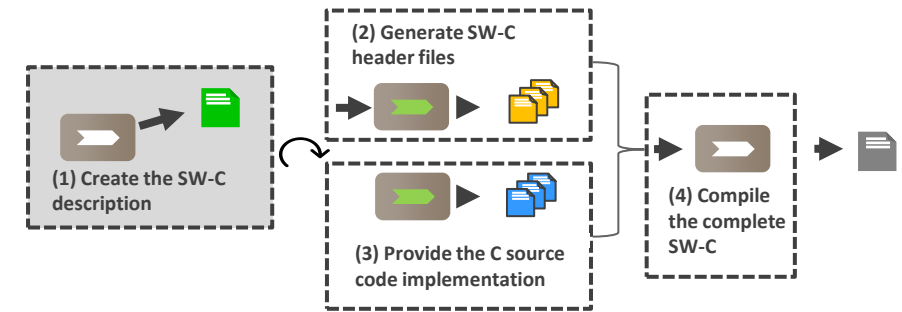


Note: *Contract phase* is used to generate the SW-C header files based on a partial AUTOSAR system (SW-C description)

(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



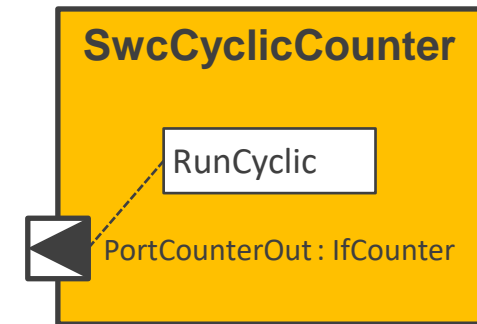
Examples in Artext will be highlighted with this green rectangle

(1) Create the SW-C description

Example SW-C: “SwcCyclicCounter”

Requirements for our SW-C (8-bit cyclic counter):

- The SW-C shall implement a counter with a period of 500ms.
- The SW-C shall provide the counter value to other SW-Cs.
- The SW-C counter value range shall be integers from 0 to 255 overflowing from 255 to 0.



Now we can design our SW-C:

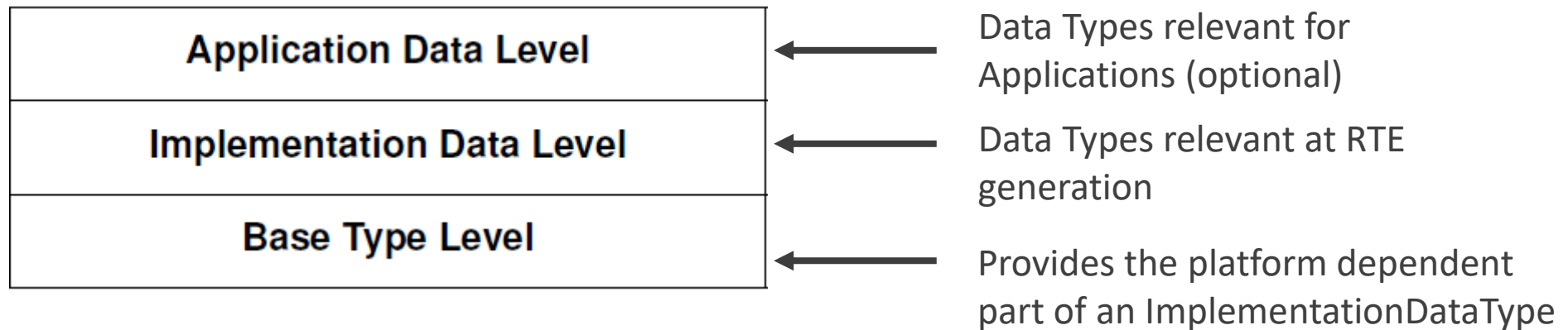
- We need a name that is a decent representation for our SW-C. → Let's call it “SwcCyclicCounter”*.
- 1 **Runnable** which shall be called every 500ms, counts and provides the value. → Let's call it “RunCyclic”.
- 1 **Port** providing the counter value to other SW-Cs. → Let's call it “PortCounterOut”.
- 1 **Interface** defining a simple 8-bit unsigned integer signal → Let's call it “IfCounter”.

* Similarities to any parts of training exercises are certainly only lucky coincidence. ;)

(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 define a simple 8-bit unsigned integer **myUInt8**:

```
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>
</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 IfCounter {  
    data myUInt8 CounterValue  
}
```

- Remember: The interface is a contract between two (or more) SW-Cs 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 SwcCyclicCounter {  
  ports {  
    sender PortCounterOut provides IfCounter  
  }  
}
```



(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 SW-C consists of one **Runnable** (function)
- The **Runnable** provides the counter value and shall be scheduled every 500ms

```
internalBehavior IbSwcCyclicCounter for SwcCyclicCounter {  
    runnable RunCyclic [0.0] {  
  
        /* Provide CounterValue via port PortCounterOut */  
        dataSendPoint PortCounterOut.CounterValue  
  
        /* Runnable period: 500ms = 0.5s */  
        timingEvent 0.5  
    }  
}
```

[0.0] is a configurable time offset for the Runnable.

The **DataSendPoint** is necessary to define that the Runnable accesses 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

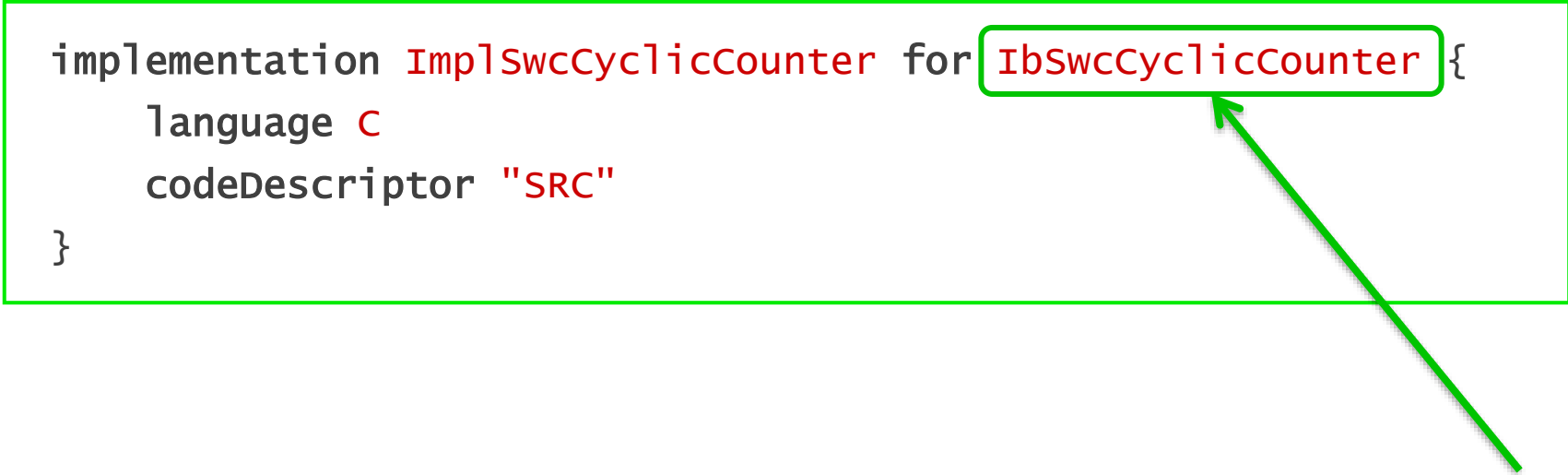


(1) Create the SW-C description

Add the SW-C Implementation description

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

```
implementation ImplSwcCyclicCounter for IbSwcCyclicCounter {  
  language C  
  codeDescriptor "SRC"  
}
```



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

(1) Create the SW-C description

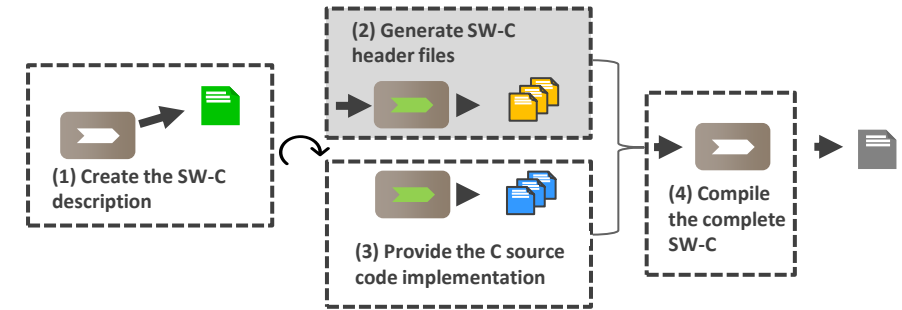
Parts of the SW-C Description - Completeness check

- 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

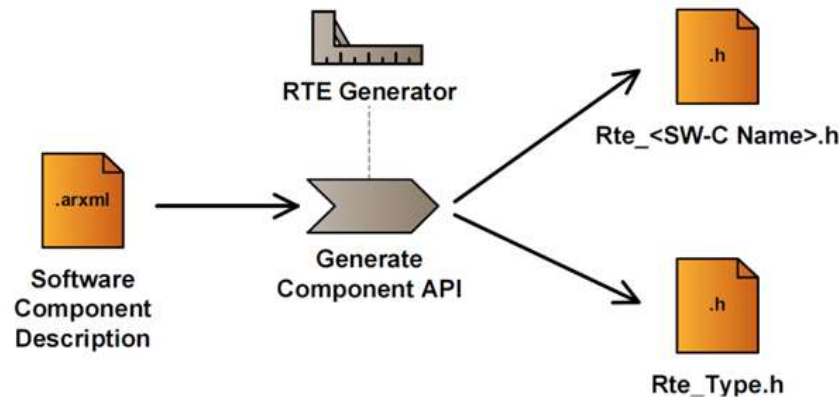


(2) Generate SW-C header files

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_SwcCyclicCounter.h*** and ***Rte_Type.h***:

Rte_SwcCyclicCounter.h (simplified):

```
#if !defined(RTE_SWCCYCLICCOUNTER_H)
#define RTE_SWCCYCLICCOUNTER_H

#include "Rte_Type.h"

/* Runnable functions */
extern void RunCyclic(void);

/* Communication APIs */
extern Std_ReturnType Rte_Write_PortCounterOut_CounterValue(myUInt8 data);

#endif /* RTE_SWCCYCLICCOUNTER_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):

```
#if !defined(RTE_TYPE_H)
#define RTE_TYPE_H

typedef uint8 myUInt8;

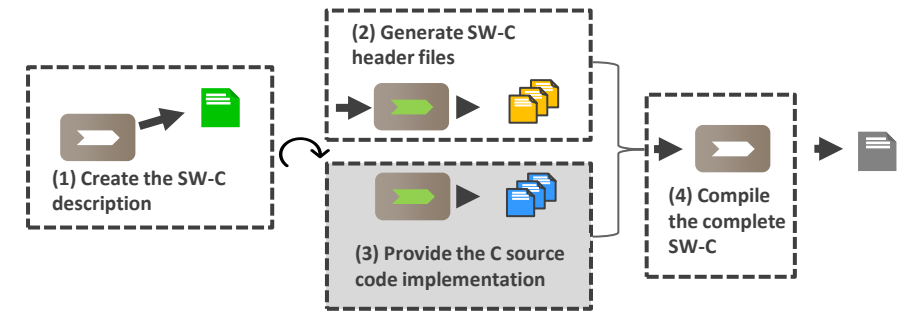
#endif /* RTE_TYPE_H */
```


(3) Implement SW-C source code

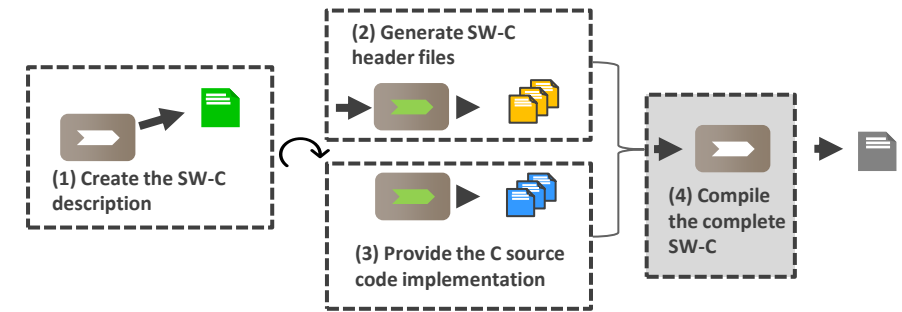
- The implementation source code for our SW-C could look like this:
- *Rte_SwcCyclicCounter.c* (example):

```
/* Include RTE APIs */
#include "Rte_SwcCyclicCounter.h"

/* Runnable "RunCyclic" will be scheduled by the RTE every 500 ms */
void RunCyclic(void)
{
    /* signal variable */
    static myUInt8 myCounter = 0;
    /* Send DataElement "CounterValue" via Port "PortCounterOut" */
    (void)Rte_Write_PortCounterOut_CounterValue(myCounter);
    myCounter++;
}
```



(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")

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 one-way communication while a **client/server** interface is used for function calls
- 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 SW-Cs and Compositions

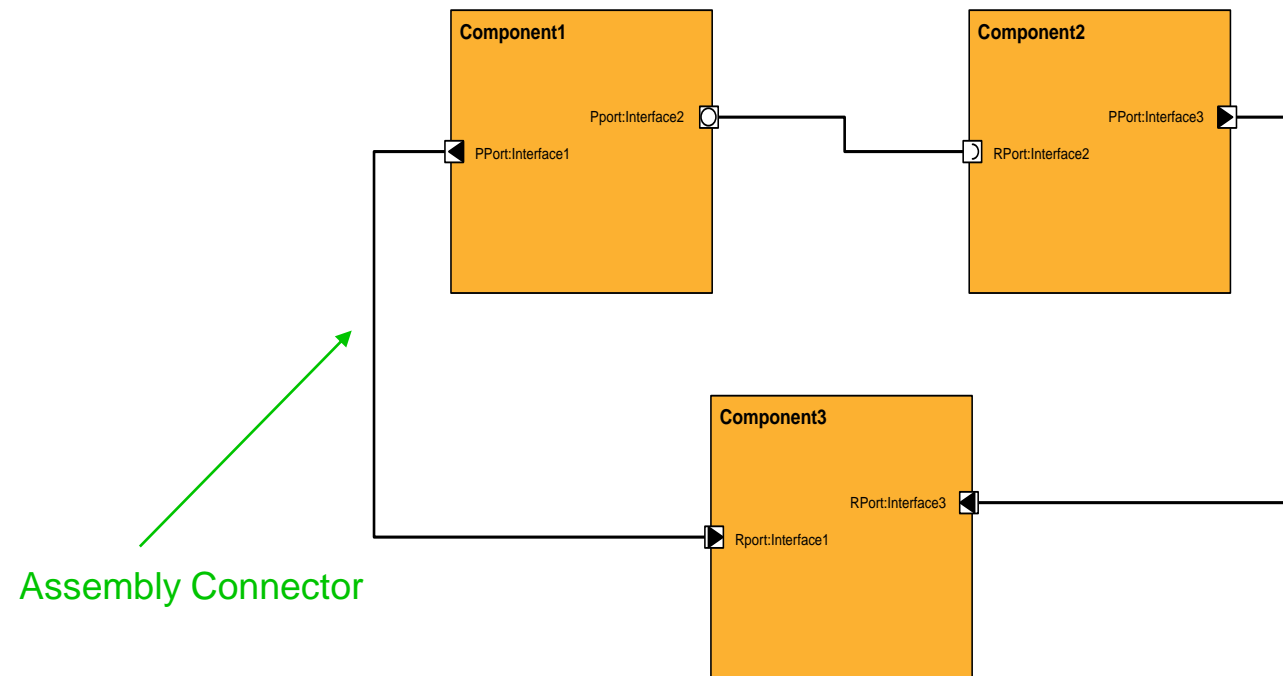


Elektrobit



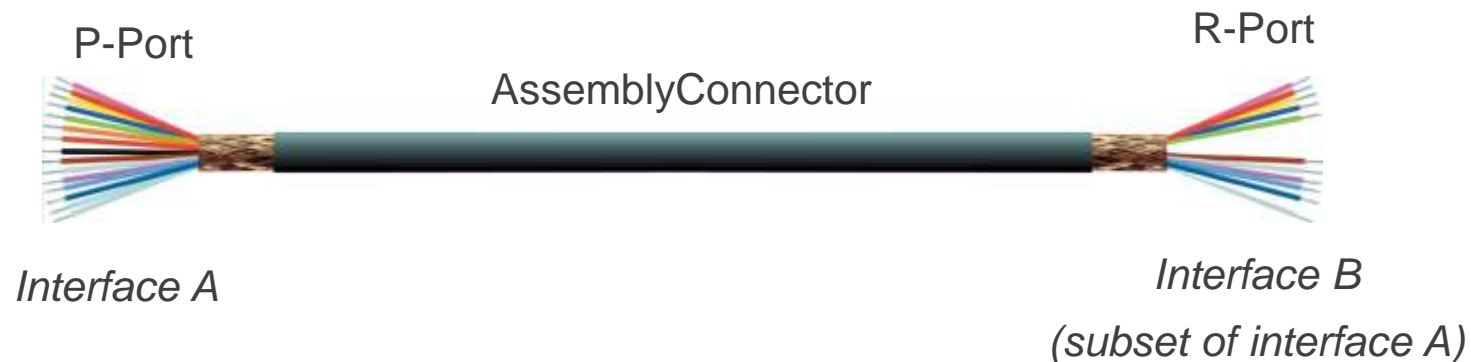
Assembly Connectors

- SW-Cs 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



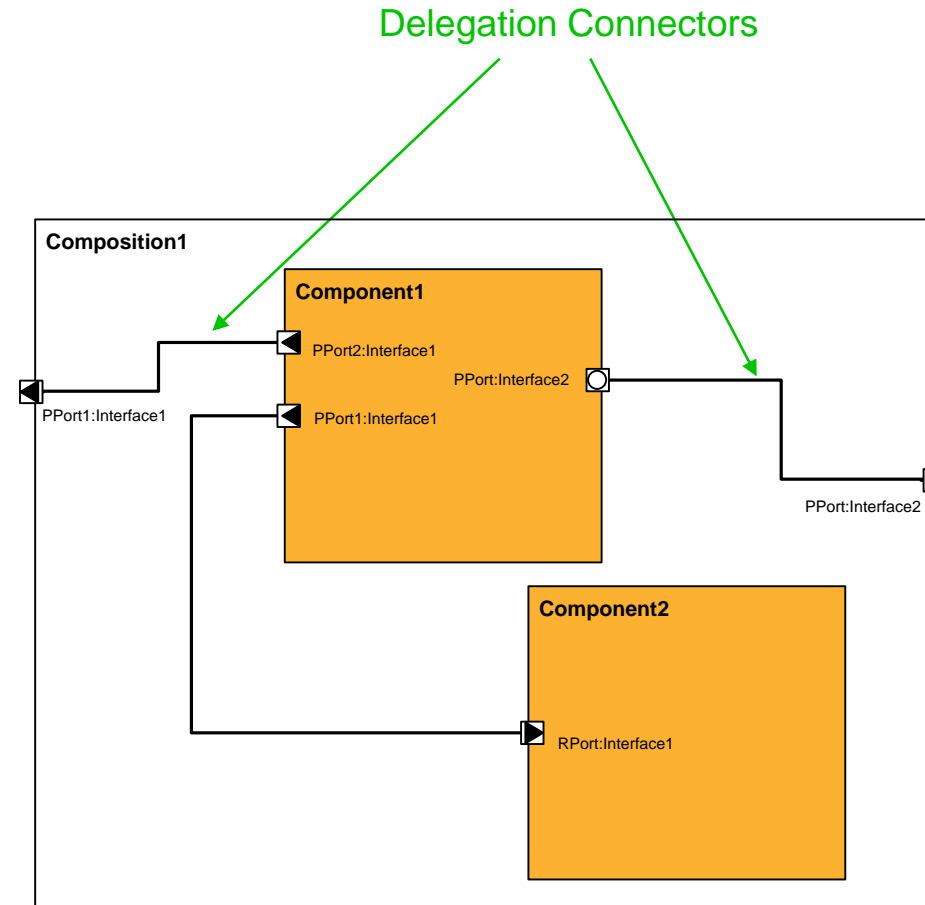
Assembly Connectors - Interface compatibility

- 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)
 - More compatibility rules for sender/receiver and client/server interfaces → next section
- The RPort's interface may be a subset of PPort's interface



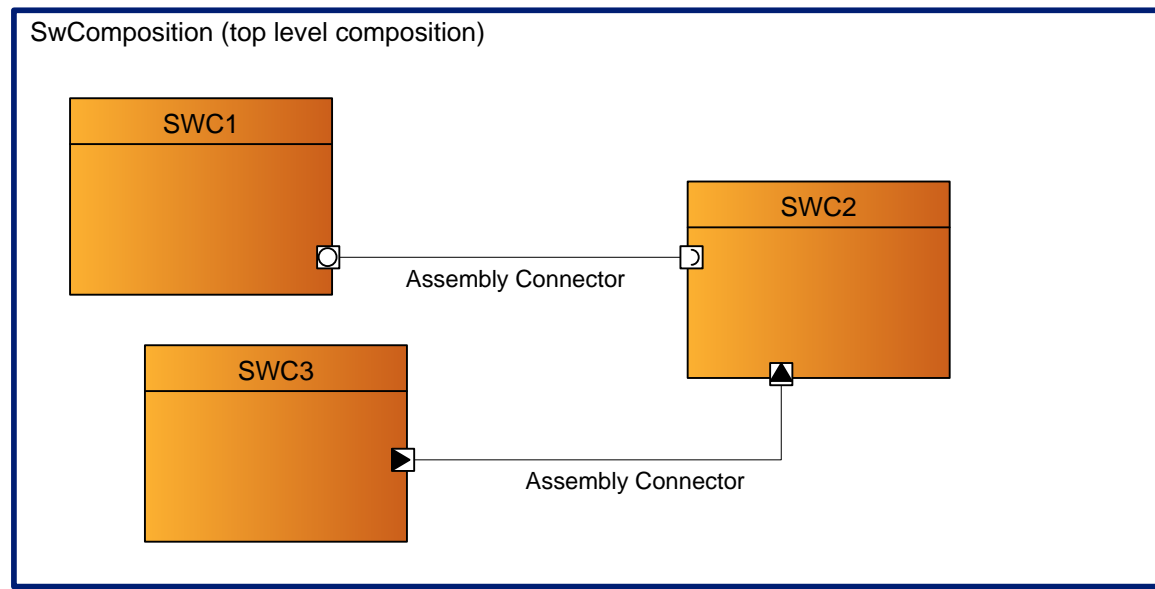
Compositions

- Several SW-Cs may be nested into a **Composition**
- A composition will act as a SW-C, 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 SW-Cs logically
 - Define interfaces between different suppliers
 - Re-arrange the content of a composition without influencing the rest of the system



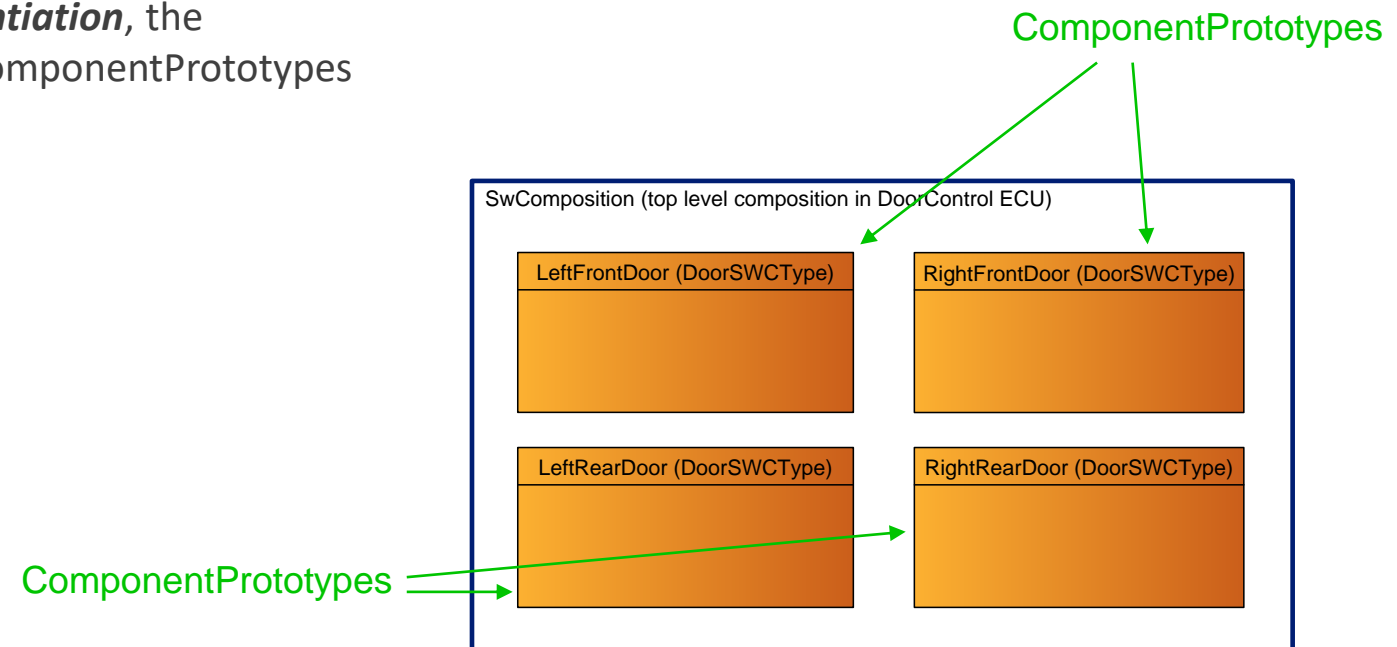
SW Composition - *The Top Level Composition*

- 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 SW-Cs and Compositions



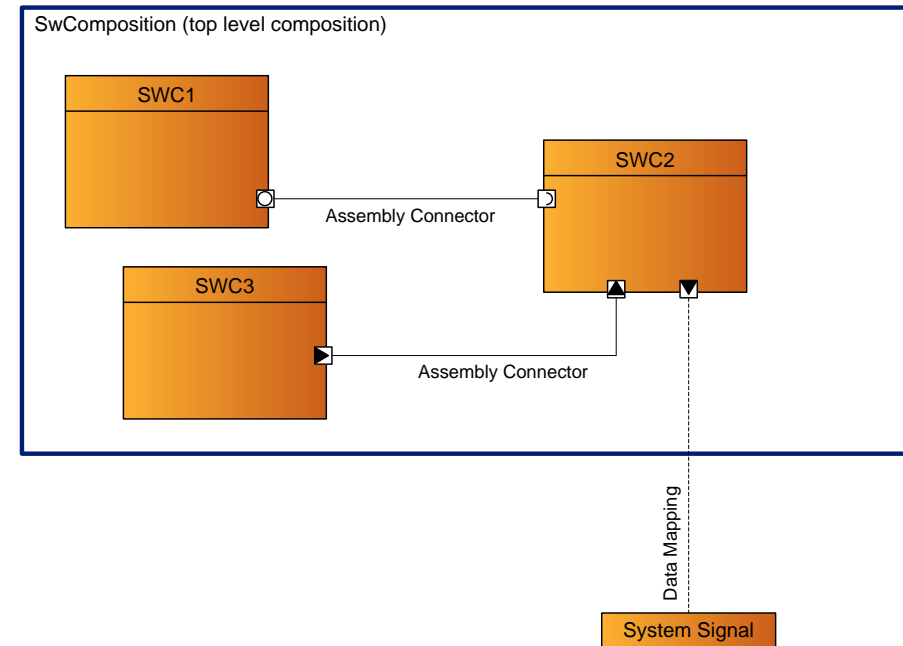
SW-C Instantiation

- 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 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 SW-Cs 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



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 Interface – 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 SW-C



Sender/Receiver Interface - Unqueued API

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

```
Rte_Write_myPPort_myDataElement(myVar);
```

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

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

Sender/Receiver Interface - Unqueued API

- Example: S/R transmission of a queued data element
(SW-C implementation code)

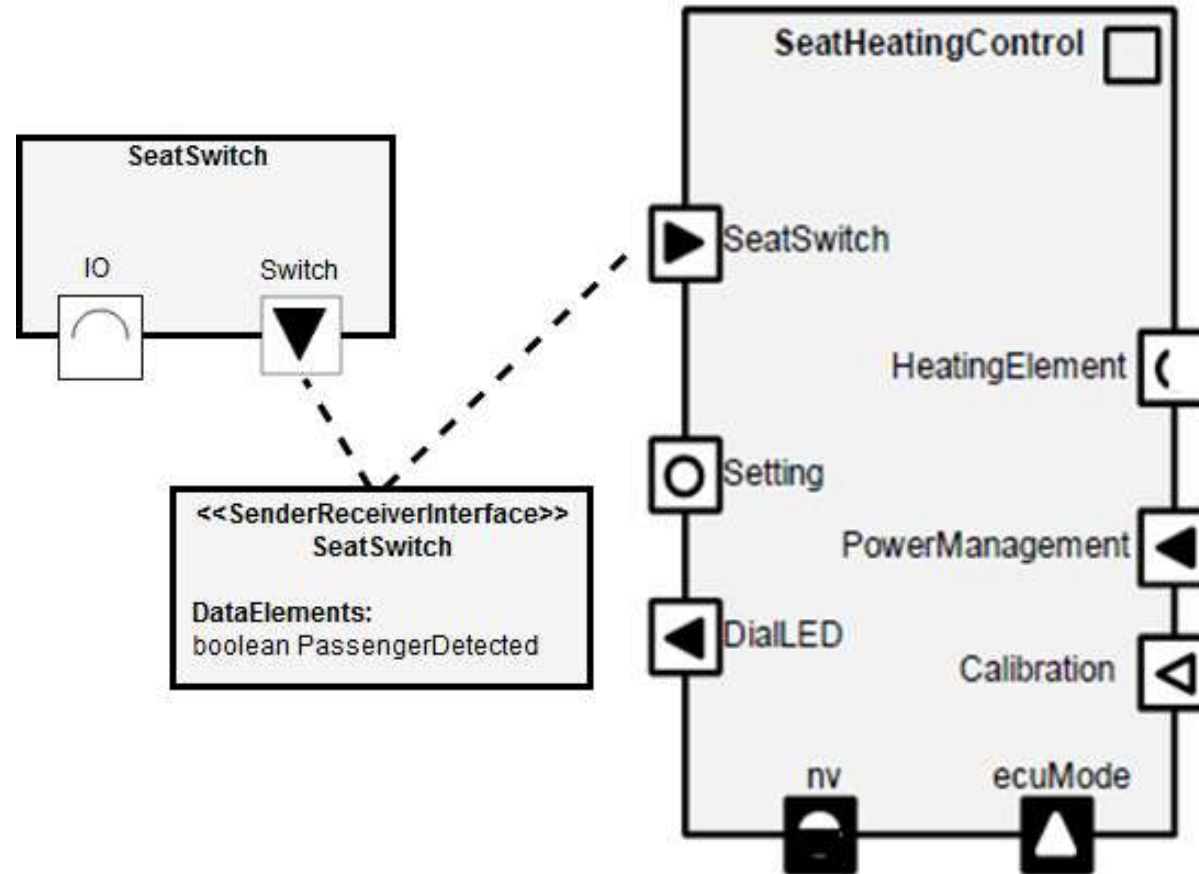
```
Rte_Send_myPPort_myDataElement(myVar);
```

- Example: S/R reception of a queued data element
(SW-C implementation code)

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

Sender/Receiver Interface – Use cases

- Concrete example:



Sender/Receiver Interface - Interface compatibility

- Two S/R interfaces are compatible if...
 - All the required DataElements in the R-Port are provided in the connected P-Port
 - the DataElements in the R-Port are a subset of the DataElements in the P-Port)
 - The DataTypes of the DataElements are compatible
 - The names of the DataElements are identical or matched by a port interface mapping

Client/Server Interface



Elektrobit



Client/Server Interface

- 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

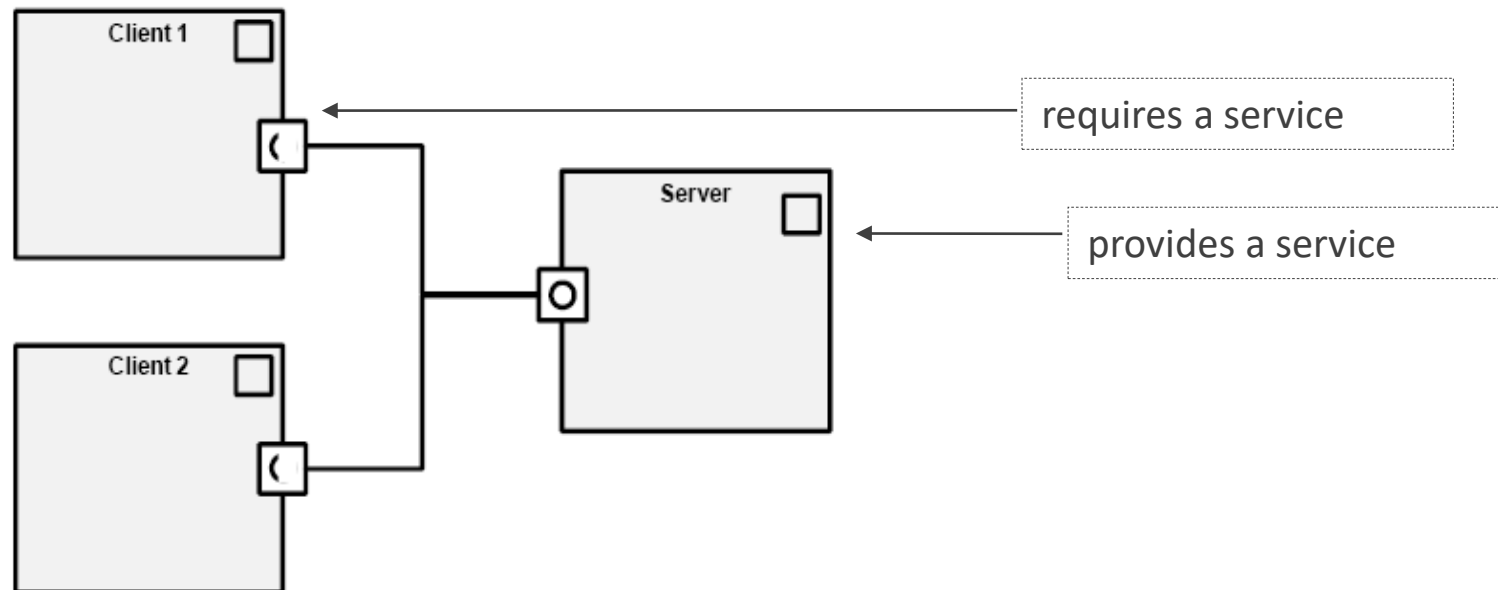
        out myUInt8 myArg2

        inout myUInt16 myArg3
    }

    operation myOperation2
    {
        ...
    }
}
```

Client/Server Interface - 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 Interface - Example

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

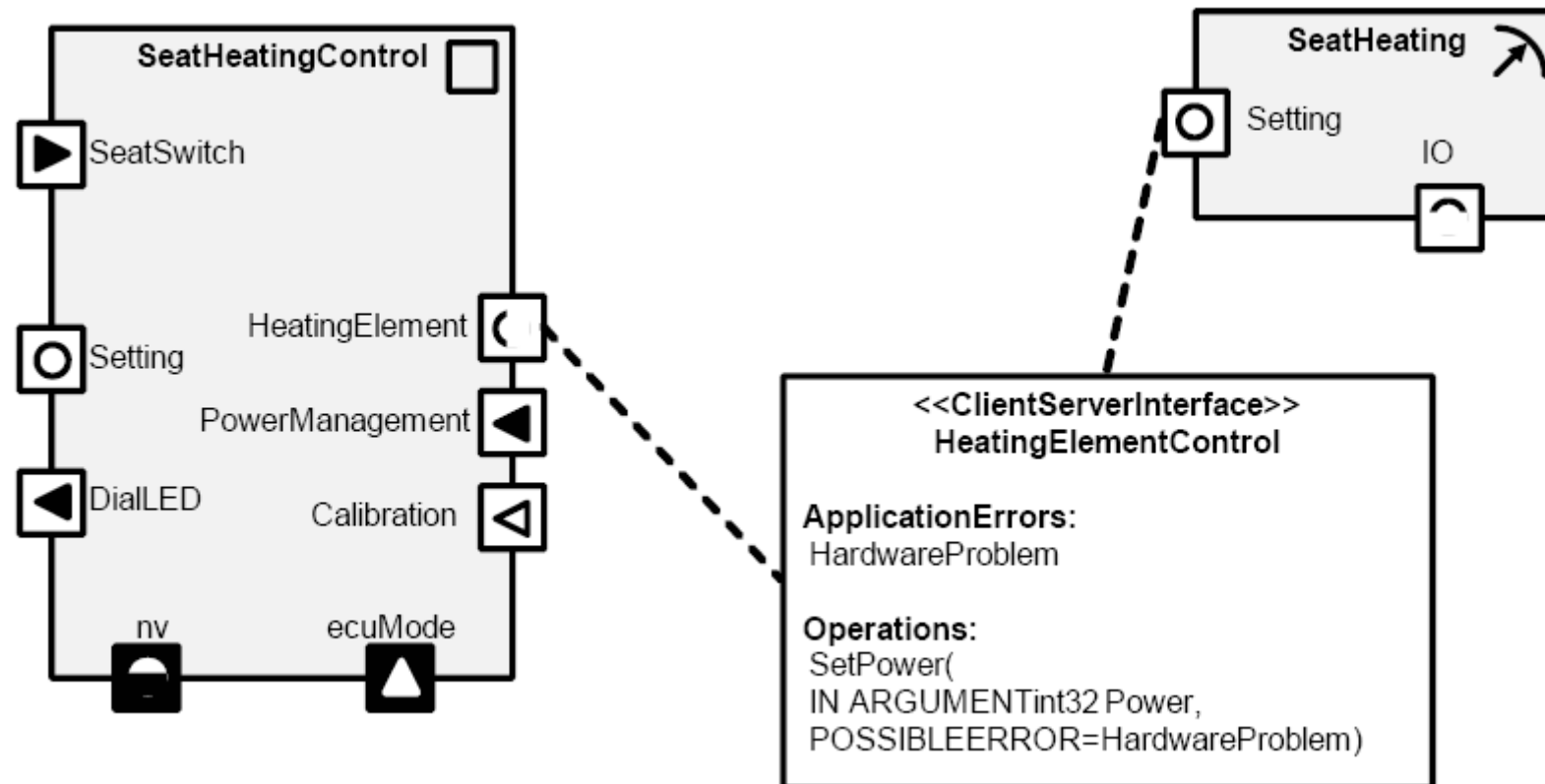
```
UInt32 myOutArg;  
Std_ReturnType myErr;  
myErr = Rte_Call_myPPort_myOperation(myInArg, &myOutArg);  
if (myErr != RTE_E_OK)  
{  
    /* Error handling here... */  
}
```

- Example: Corresponding server implementation

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

Client/Server Interface - Use cases

- Concrete example:



Client/Server Interface - Interface compatibility

- Two Client/Server 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 or matched by a port interface mapping
 - The Arguments are compatible (same name, same direction, data types compatible)





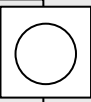
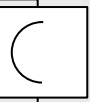


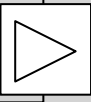
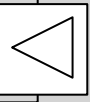






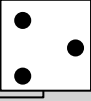
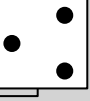
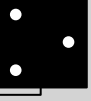
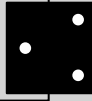




Other types of interfaces



Elektrobit



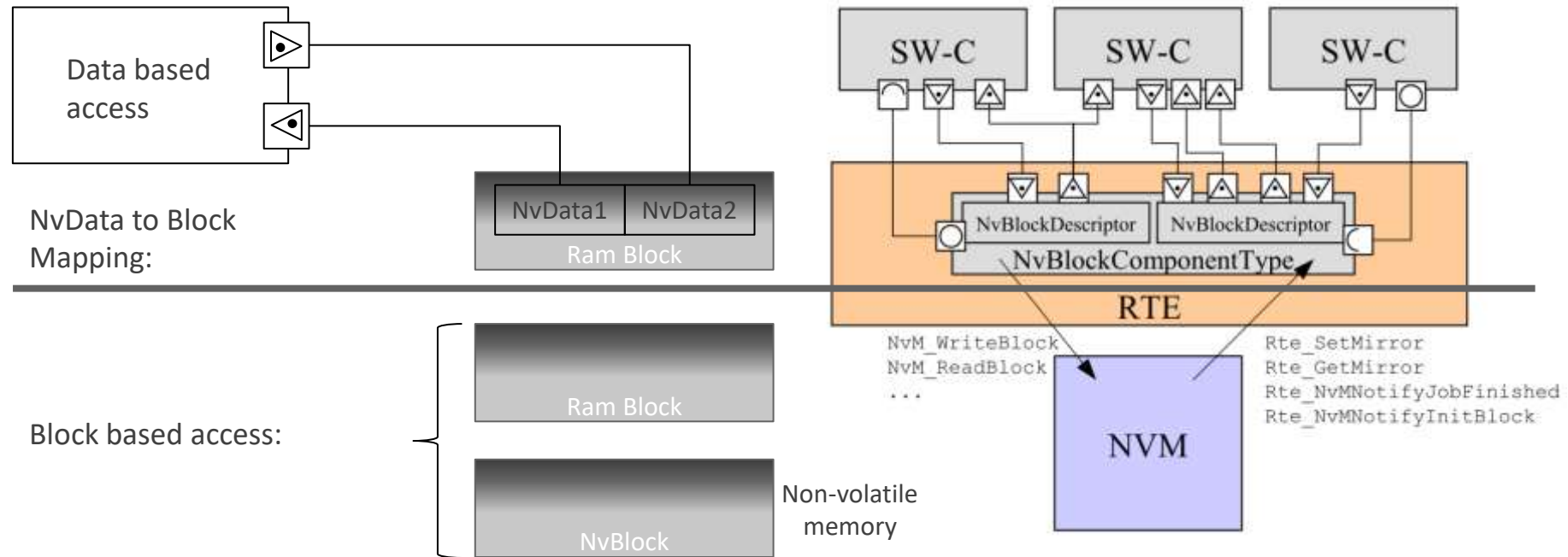
Port icons for different kind of interfaces

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 data 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 (SW-C)

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

The Runtime Environment (RTE)



Elektrobit



Section Overview – The Runtime Environment

Software Components

- AUTOSAR Software Component Description
- Basic Elements of an SW-C
- How to create a SW-C
- Connecting SW-Cs 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
- Partitioning

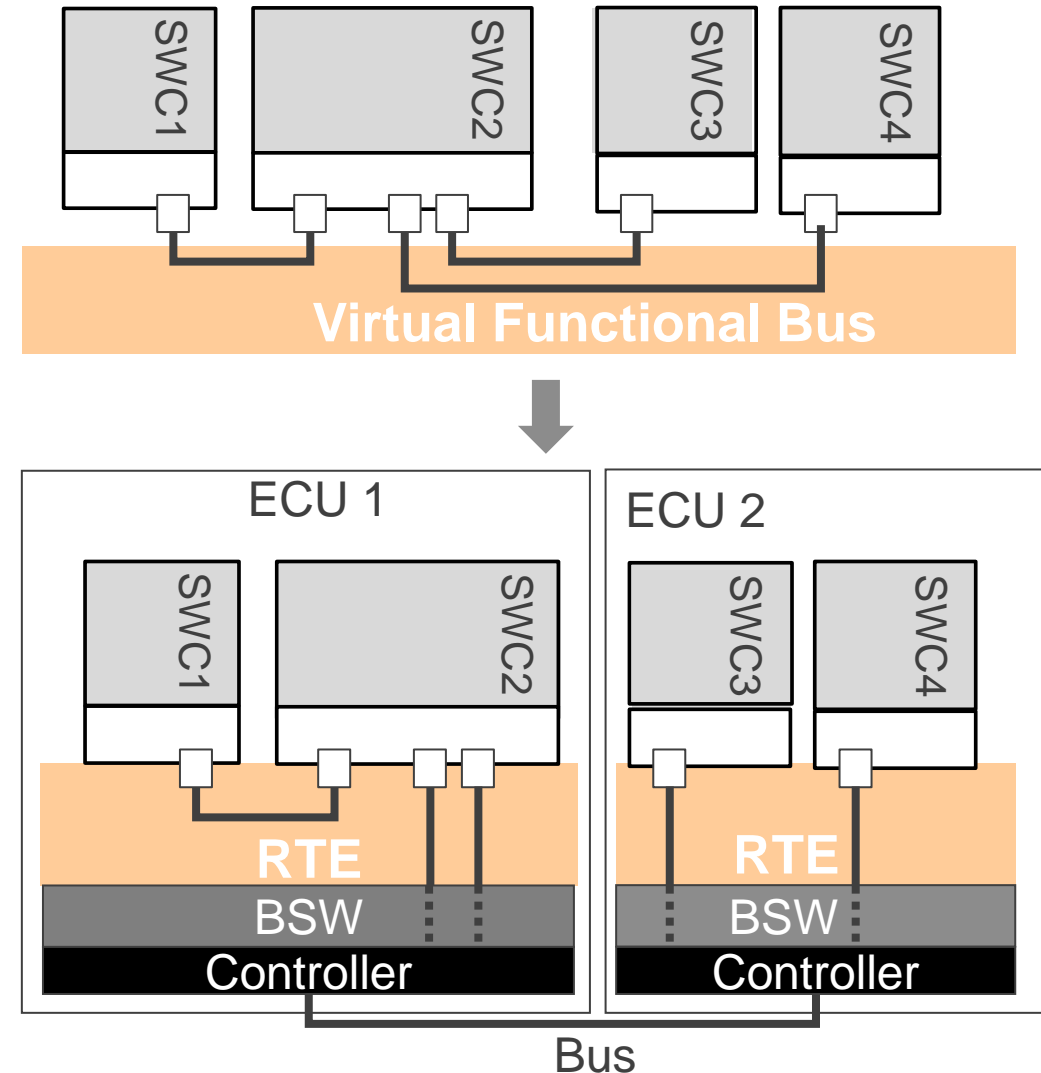
Advanced SW-C Concepts

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

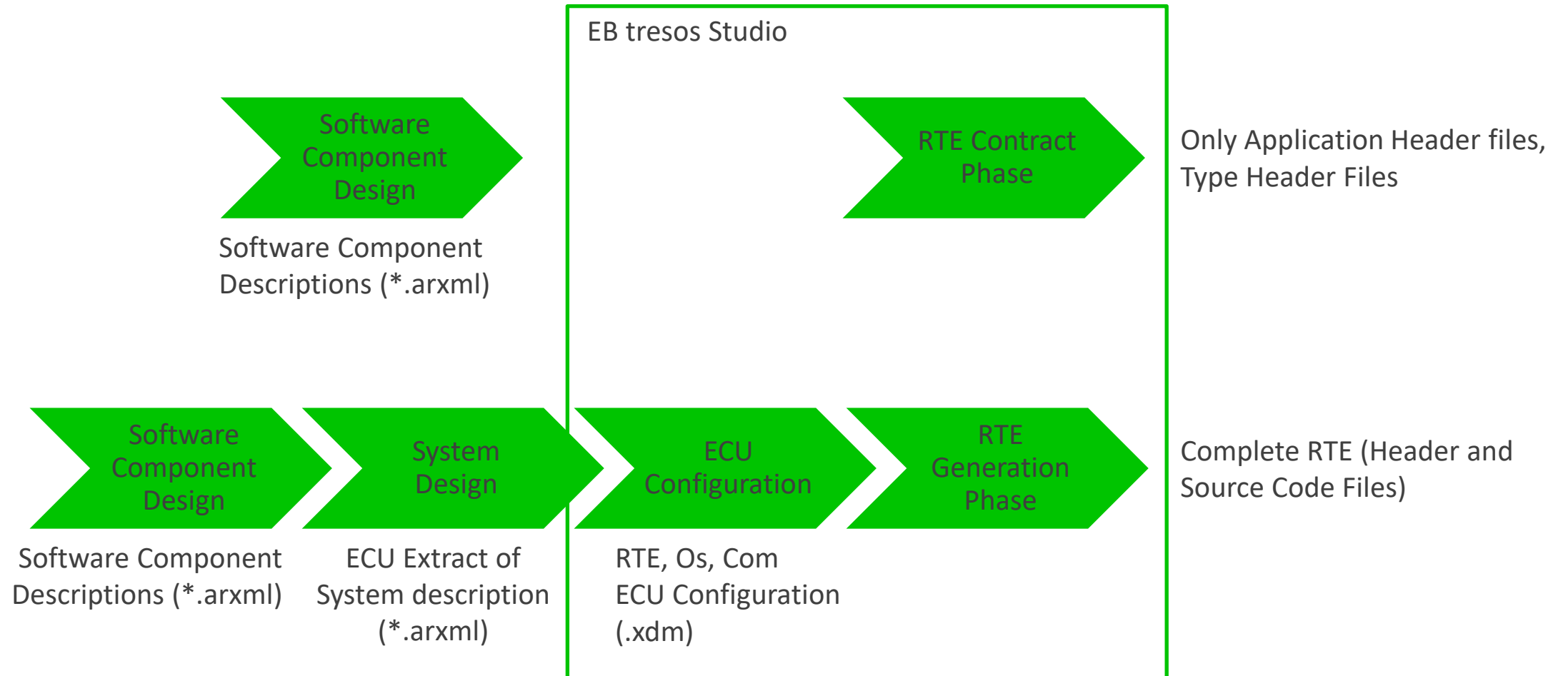
The Runtime Environment (RTE)

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

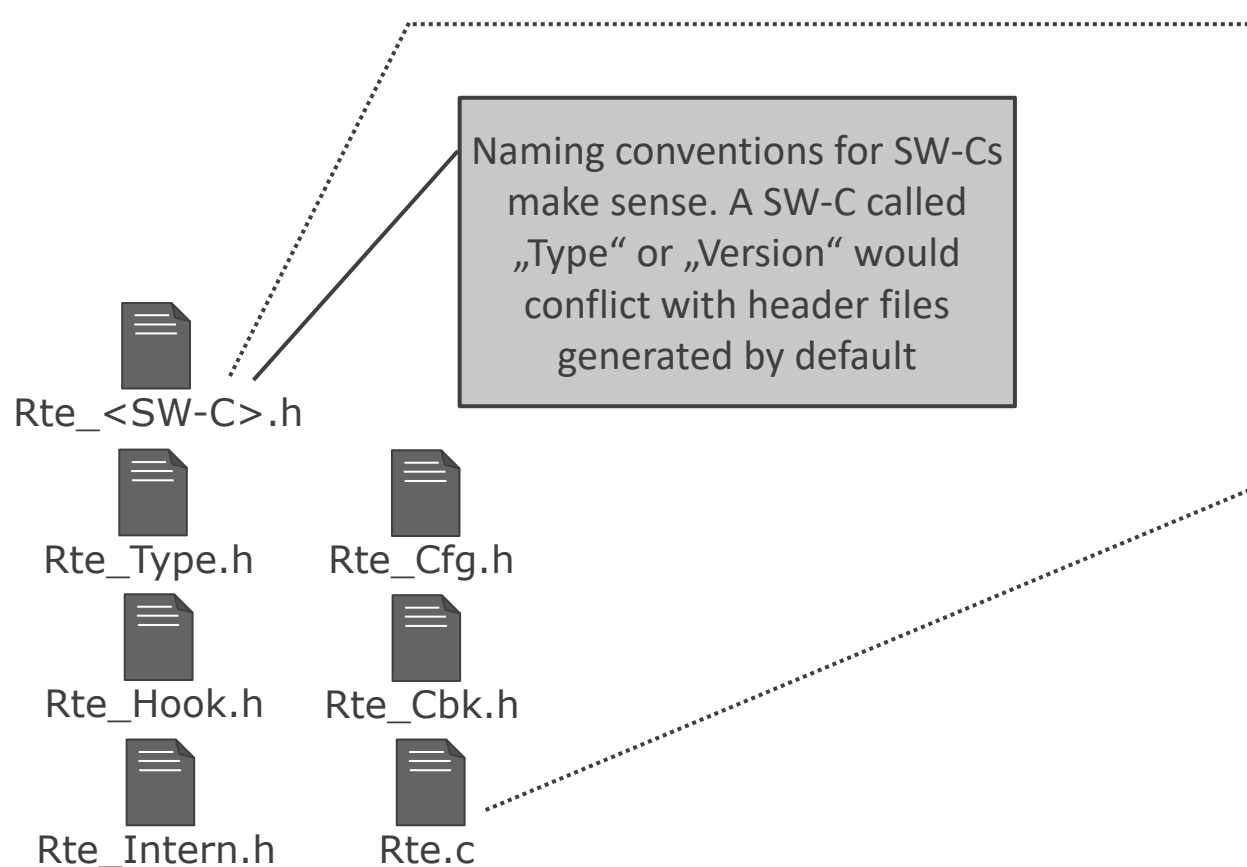


Workflow – RTE Generation



RTE Generation Phase - Output files

- Generated files: Entire compilable RTE source code and additional information



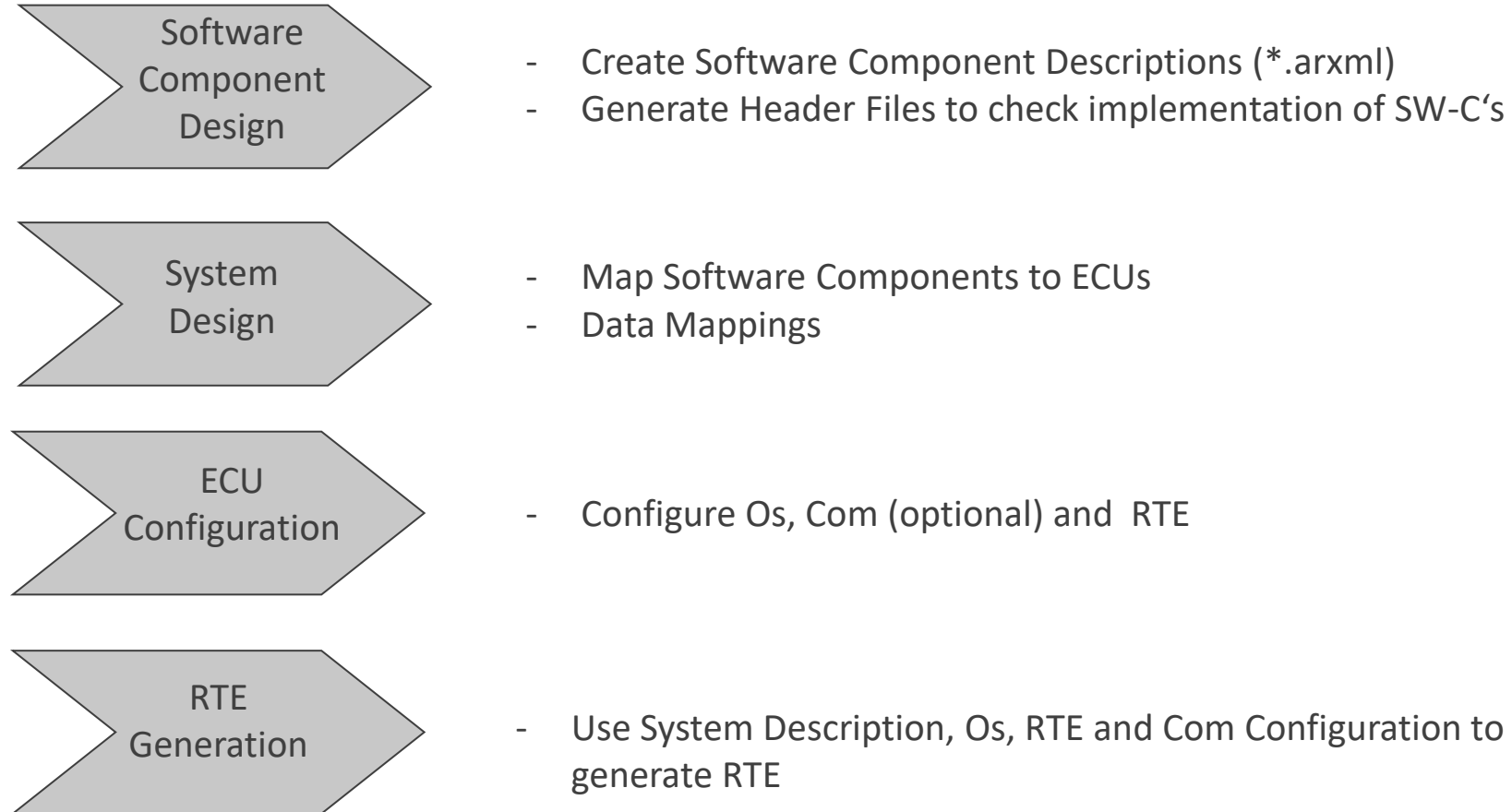
Application Header Files

- 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

Summary: Workflow for RTE Generation Process



RTE events and event mapping



Elektrobit

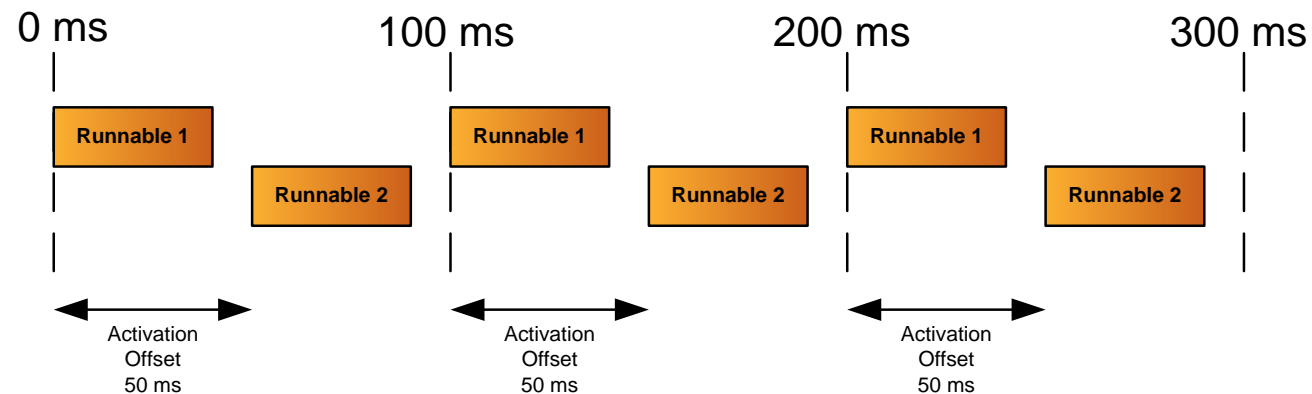


Event overview

- 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**

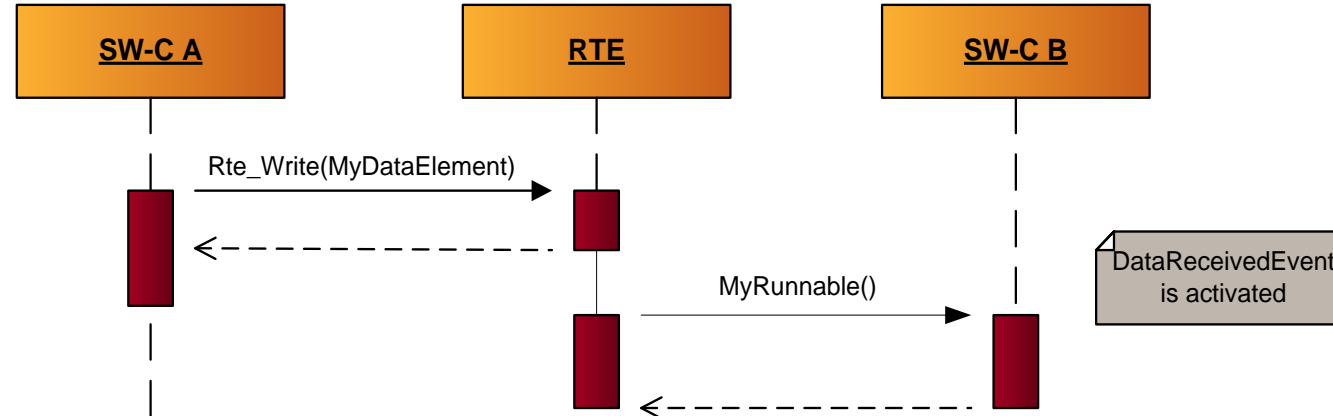
Timing event

- 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)

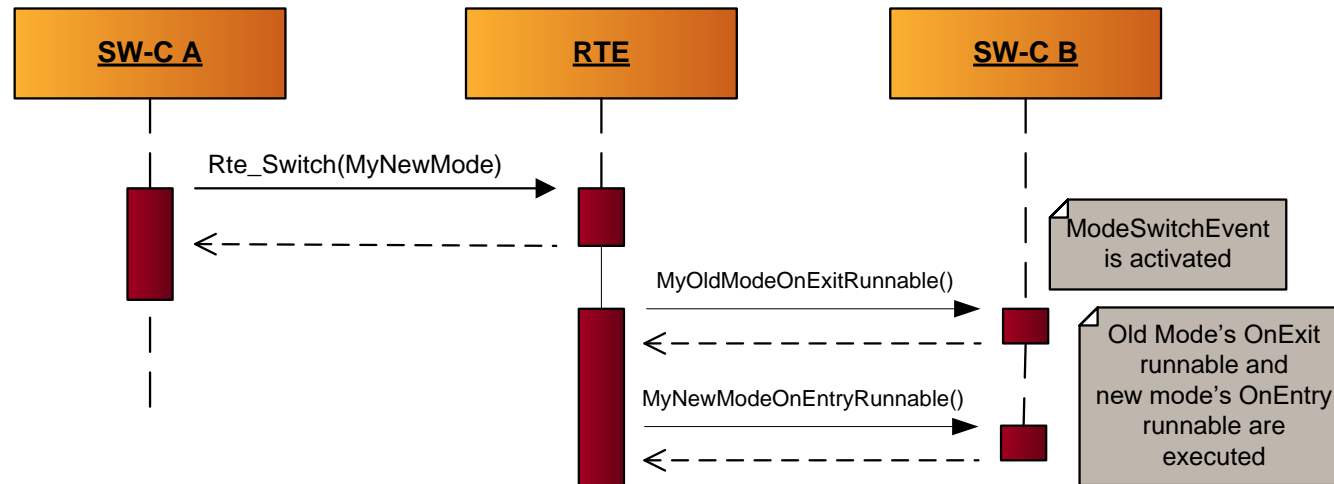
- A **Data Received Event** connects a certain DataElement in a port to a Runnable
- Upon data reception, the RTE triggers the Runnable as soon as possible
- If many DataReceivedEvents are connected to the same Runnable, it is possible to determine which one triggered the execution if the **activating 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)

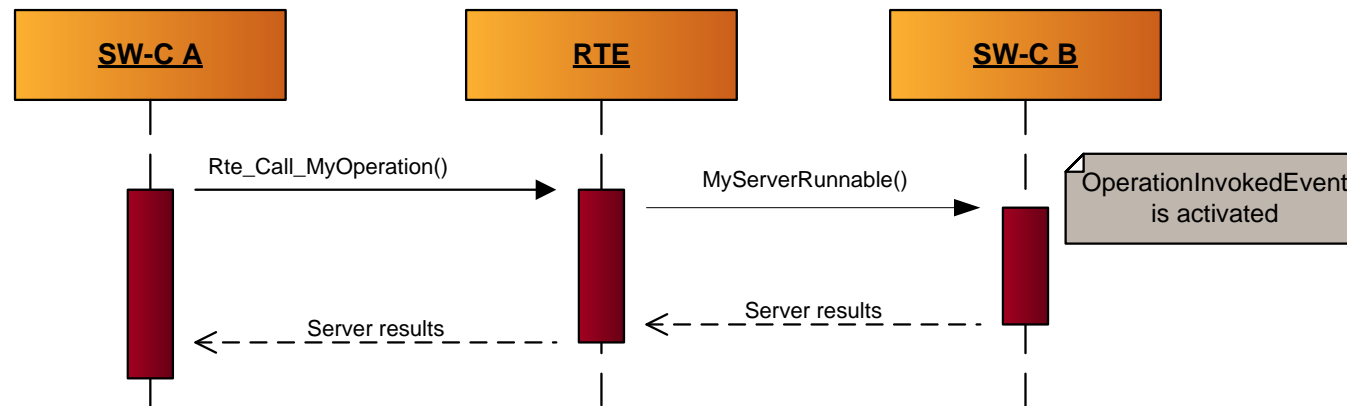
Mode switched events

- 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:

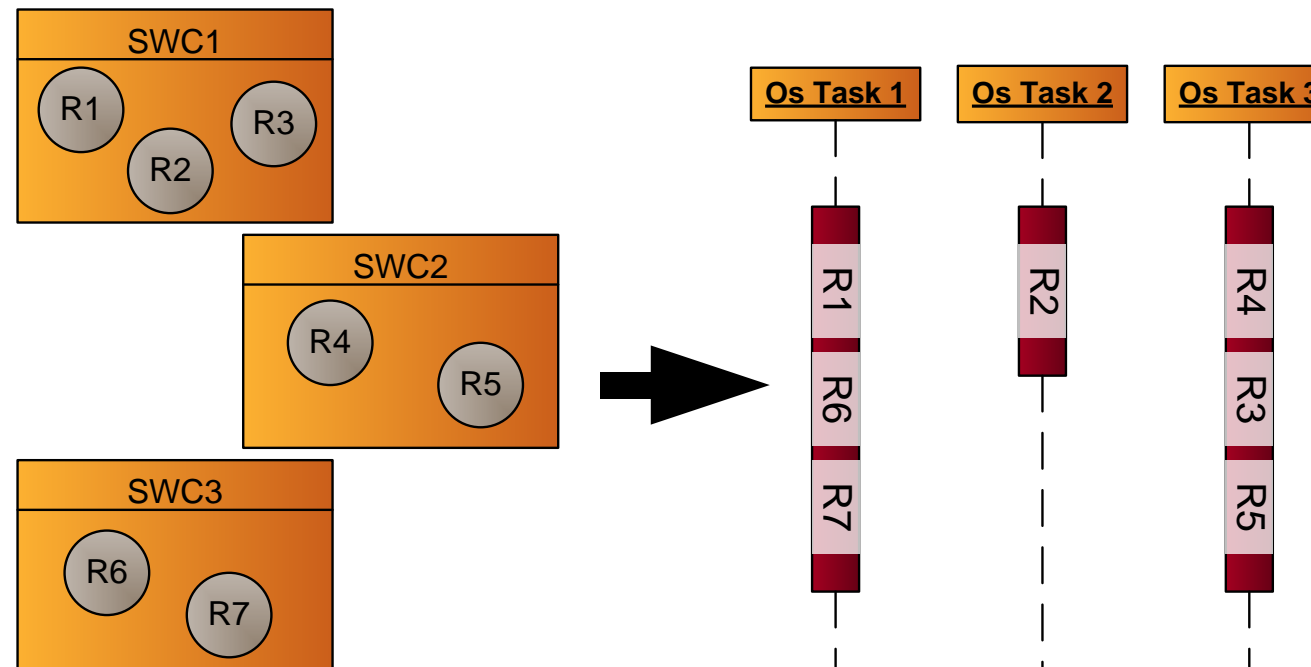


Runnable Entity Mappings - Assigning events to Os tasks

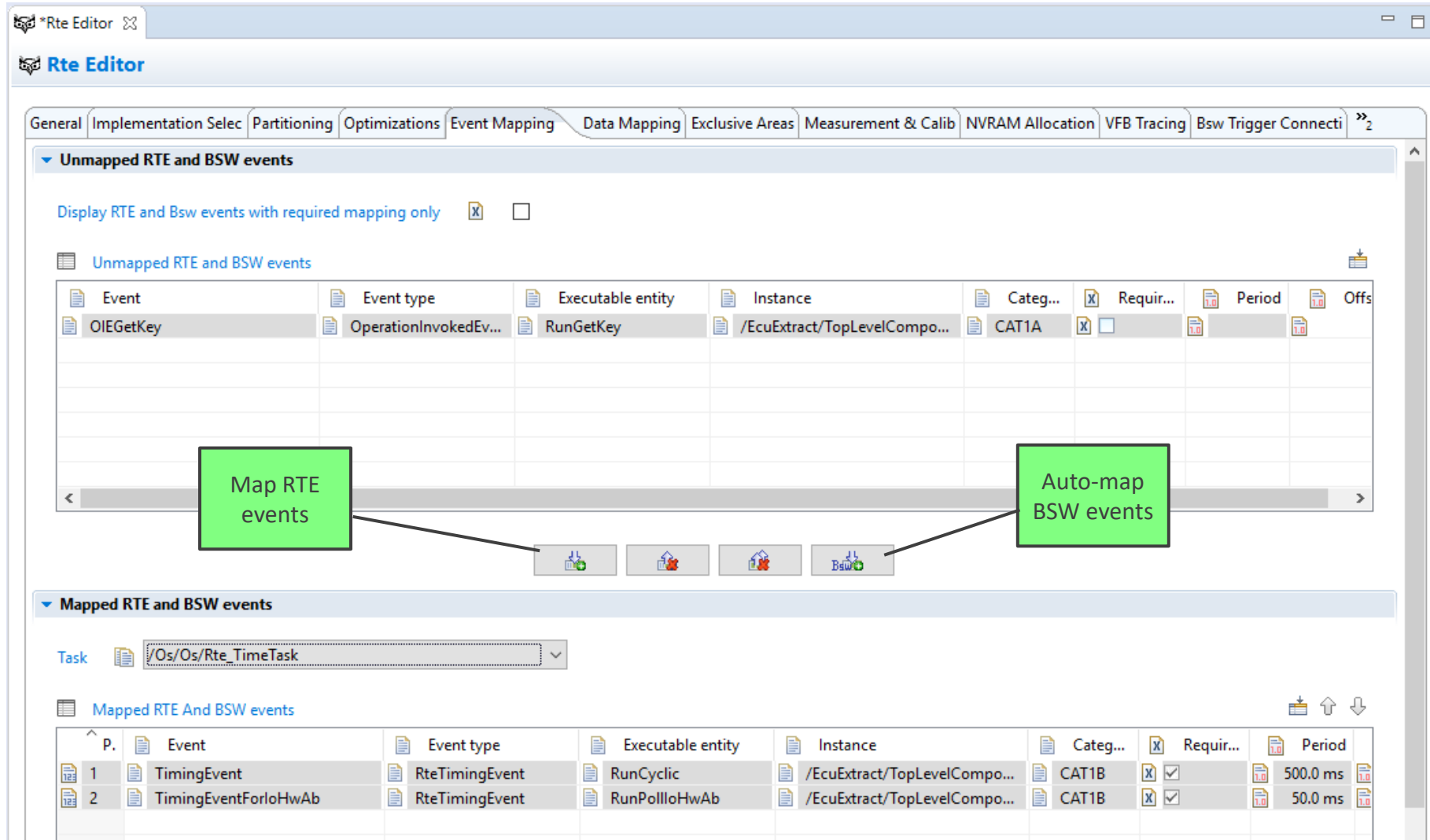
- Multiple RteEvents can trigger the same runnable
 - **Example:** A TimingEvent and a DataReceivedEvent can trigger the same runnable
- The RteEvents (and implicitly the Runnables) must be **mapped** to Os tasks – so called **Runnable Entity Mapping**
- Several RteEvents (Runnables) can be mapped to 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



Runnable Entity mapping – example in EB tresos Studio



Unmapped RTE and BSW events

Display RTE and Bsw events with required mapping only ☒ ☐

Event	Event type	Executable entity	Instance	Categ...	Requir...	Period	Offs
OIEGetKey	OperationInvokedEv...	RunGetKey	/EcuExtract/TopLevelCompo...	CAT1A	<input checked="" type="checkbox"/>		

Map RTE events **Auto-map BSW events**

Mapped RTE and BSW events

Task

P.	Event	Event type	Executable entity	Instance	Categ...	Requir...	Period
1	TimingEvent	RteTimingEvent	RunCyclic	/EcuExtract/TopLevelCompo...	CAT1B	<input checked="" type="checkbox"/>	500.0 ms
2	TimingEventForIoHwAb	RteTimingEvent	RunPollioHwAb	/EcuExtract/TopLevelCompo...	CAT1B	<input checked="" type="checkbox"/>	50.0 ms

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

RTE Partitioning

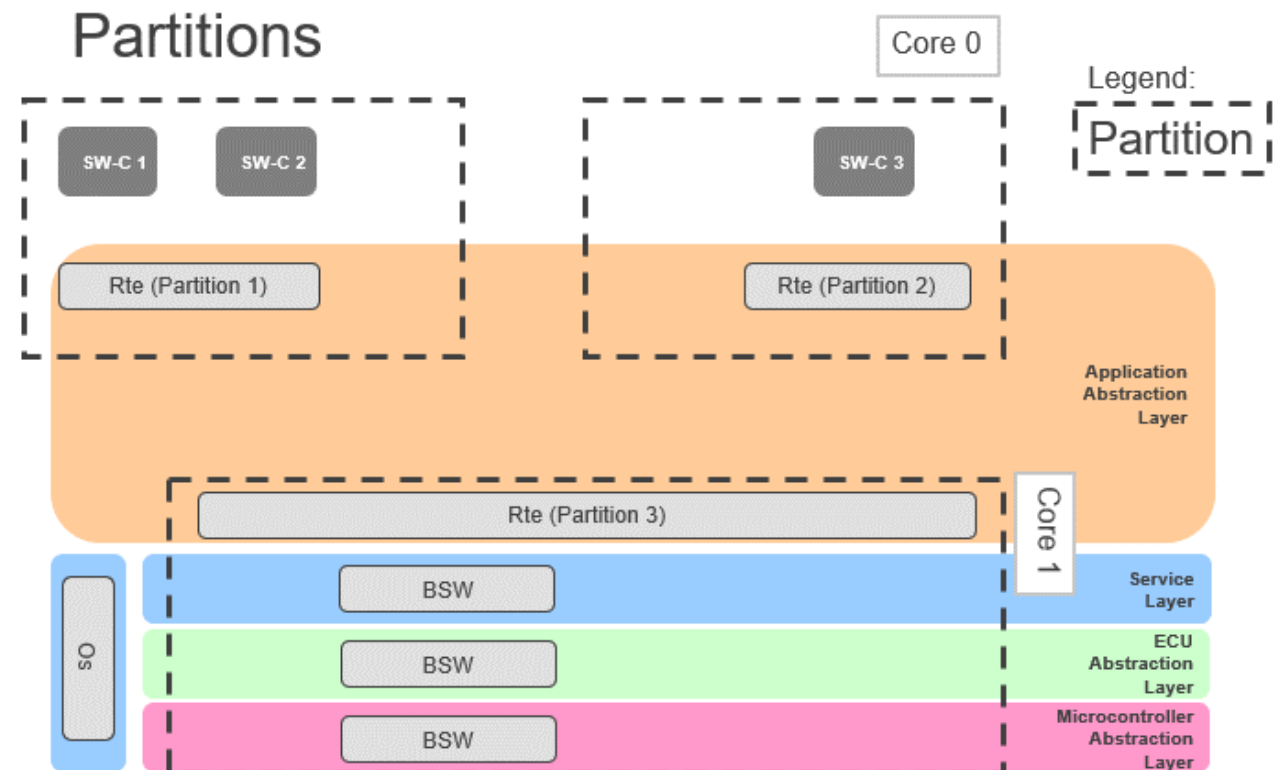


Elektrobit



Introduction

- AUTOSAR allows to partition the ECU software. Use cases for **partitioning** are:
 - Multi-core systems
 - Systems that enable memory protection
- Partitions are based on **OS-Applications**.
 - Each OS-Application represents one partition. Therefore ...
 - Each partition is assigned to exactly one core.
 - Several partitions can be assigned to the same core.
- A **separate** RTE is generated for each partition.



Inter-Partition Communication

- **Inter-Partition Communication** describes communication between different partitions. Other communication forms are **Intra-Partition Communication** (i.e. communication inside a partition) and **Inter-ECU Communication** (i.e. communication to a different ECU).
- SW-Cs do not need to care about the communication type and partition mapping:
 - Communication APIs are not affected by partitioning.
 - The RTE must take care of **data integrity** and **synchronization** for communication (this includes communication between different partitions). Note however that protection mechanisms are typically configurable so make sure to not invalidate the protection by an inappropriate configuration.
 - The RTE must support mixed communication scenarios (e.g. a 1:n S/R communication where the sender in one partition has receivers in the same as well as in different partitions).
- The underlying mechanism for Inter-Partition Communication is implementation- and configuration dependent:
 - AUTOSAR specifies the **Inter-OS-Application Communicator (IOC)** as part of the Os.
 - EB provides an RTE internal mechanism called **Shared Memory Communicator (SMC)**.
- Note: Communication as well as synchronization between different partitions can be more expensive than staying within one partition, especially if crossing core boundaries. Therefore partition your ECU with care.

Advanced SW-C concepts



Elektrobit



Section overview – Advanced SW-Cs Concepts

Software Components

- AUTOSAR Software Component Description
- Basic Elements of an SW-C
- How to create a SW-C
- Connecting SW-Cs 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
- Partitioning

Advanced SW-C Concepts

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

Advanced SW-C concepts Sender/Receiver Interfaces

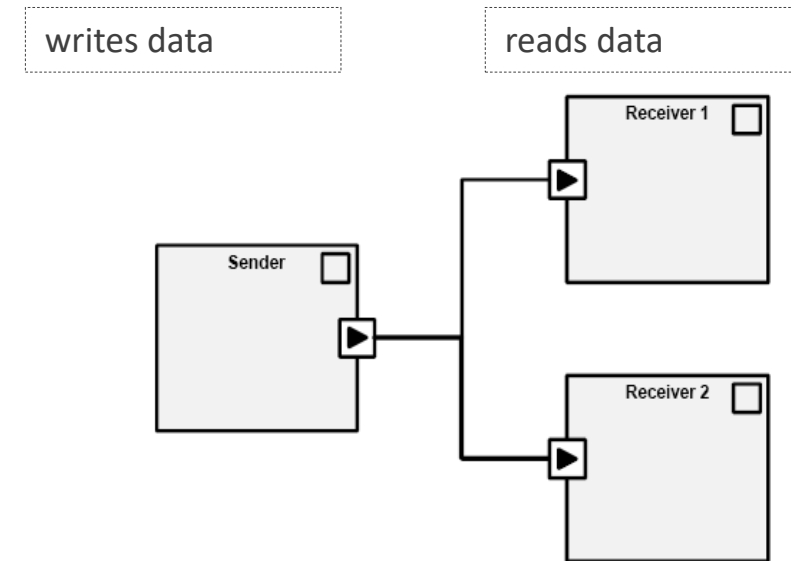


Elektrobit



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***



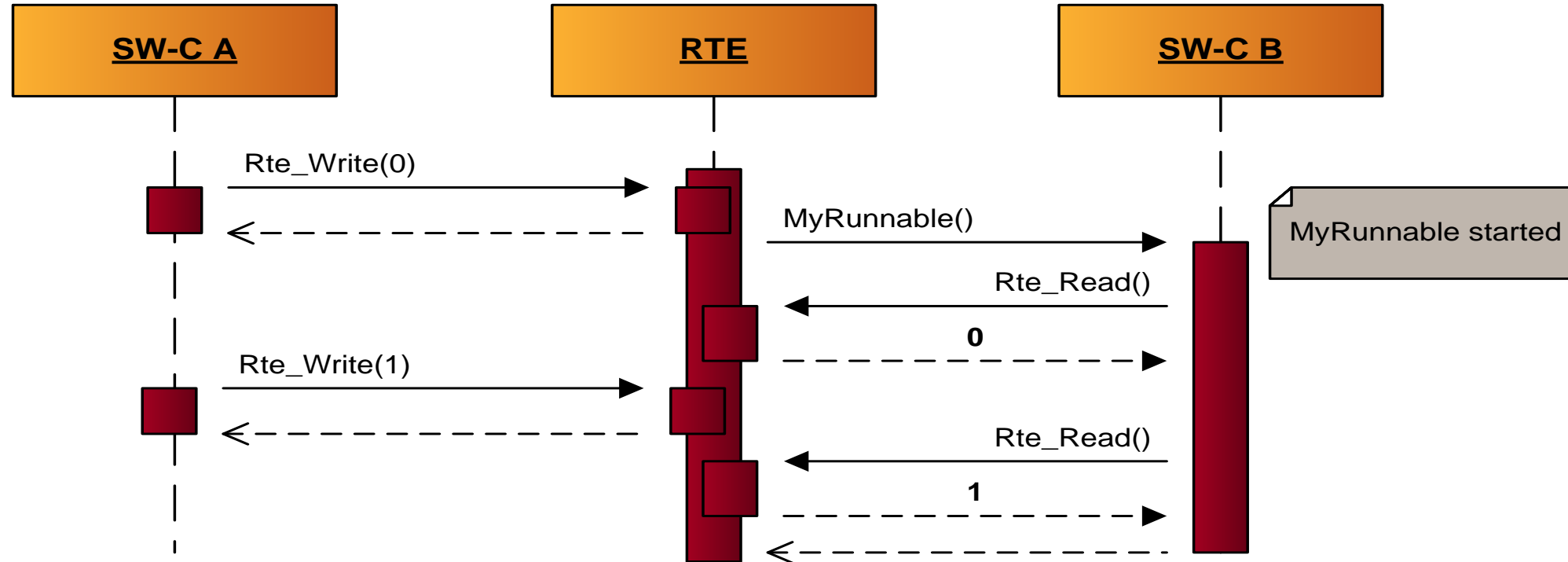
Explicit vs. Implicit communication

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

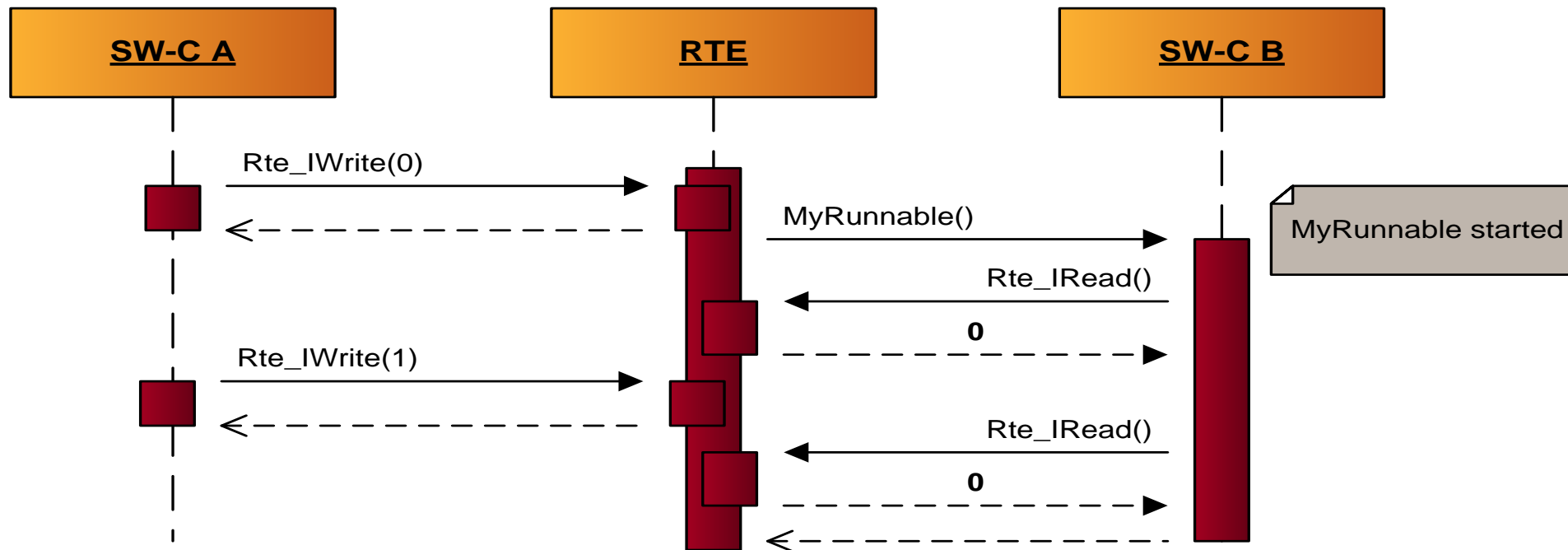
- When receiving queued data, you must check the return value:

```
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 unqueued 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 unqueued data:

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

NOTE: Data was read when Runnable function started



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

Blocking vs. Non-blocking

- **Blocking** APIs suspend their execution until a defined condition is met (e.g. “queue is not empty” for a blocking `Rte_Receive()`). If that condition is met they perform their task and return to the caller.
- **Non-blocking** APIs perform their task and return to the caller. They do not suspend their execution.
- Depending on the configuration, the RTE API functions for explicit queued **reception** (`Rte_Receive()`) can be
 - **Blocking**: API waits for new data to arrive (if the queue is empty) or for a time-out to expire
 - **Non-Blocking**: API returns immediately:
 - 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).
- Other sender/receiver related RTE APIs are non-blocking:
 - `Rte_Write()/Rte_Send()` return immediately after the transmission was initiated.
 - `Rte_Read()` returns immediately after the reception.
 - `Rte_IWrite()/Rte_IRead()` return immediately after the buffer was written/read.

Blocking Rte_Receive - Usage

- The code for calling a blocking Rte_Receive() API looks 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 results in an **extended 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 API (Rte_Receive()) can be set as **blocking** by defining a **WaitPoint** in combination with a **DataReceivedEvent**.

Advanced SW-C 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***

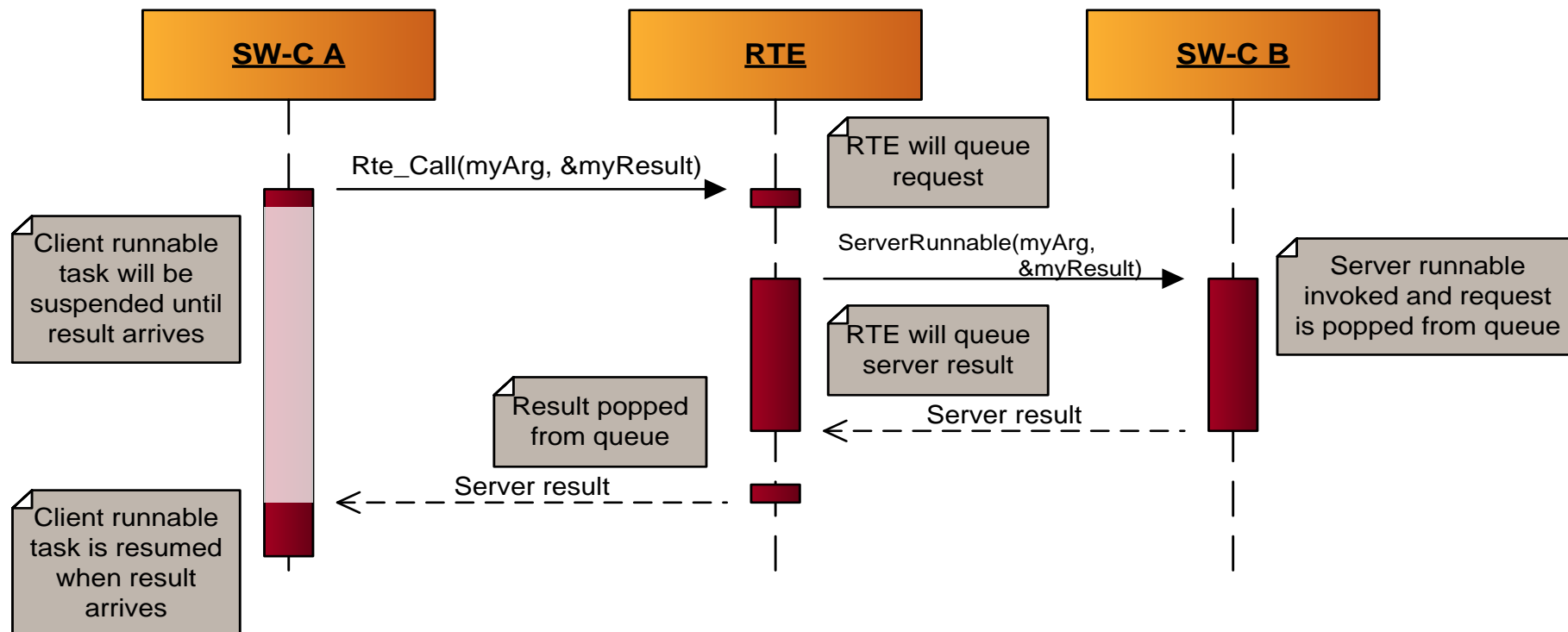
- The client software component calls an operation synchronously (Rte_Call())
- The RTE API call generated for the call **blocks**
 - until the server execution has finished and the results are available, or
 - until an error occurs.
- A special case of synchronous call is the direct call

- ***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

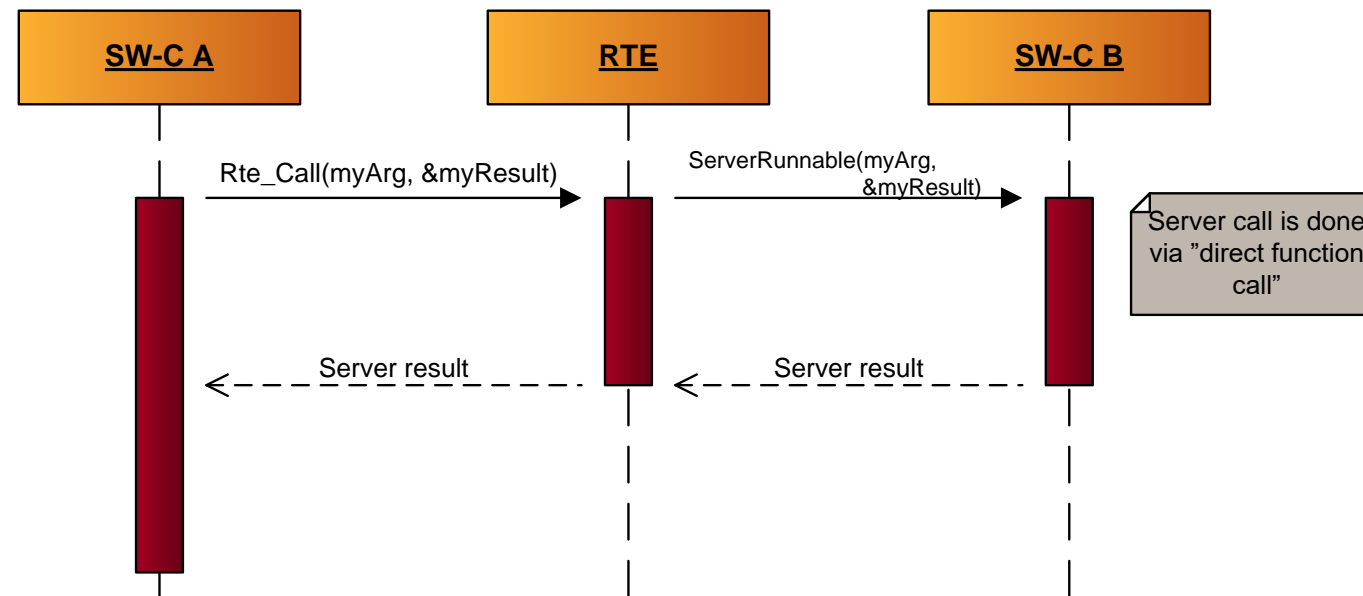
Synchronous Server Call - *Basic case*

- 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 SW-C concepts

Interrunnable variables



Elektrobit



INTERRUNNABLE VARIABLES - Configuration

- ***Interrunnable variables*** are used to share data between runnables of the same SW-C
- 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 SW-C concepts

SW-C Instantiation

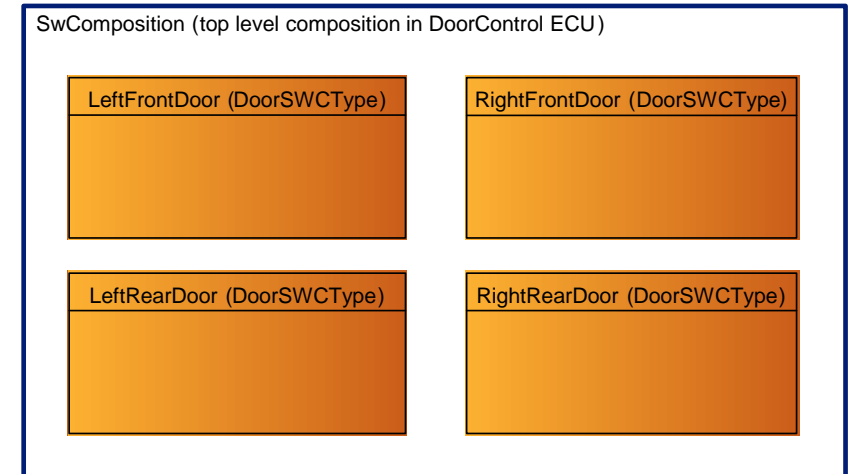


Elektrobit



SW-C Instantiation – Quick review

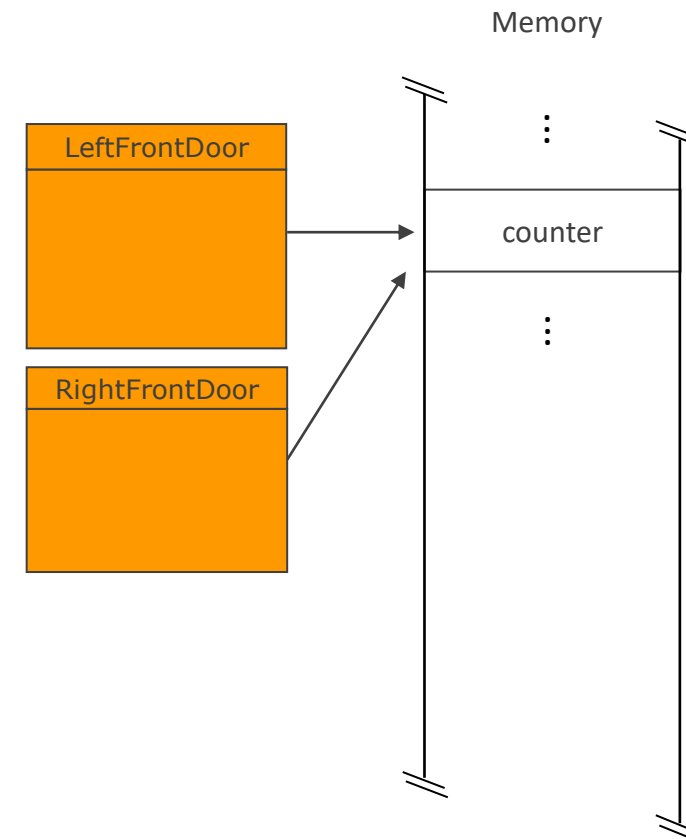
- A component **instance** in a composition is called a **ComponentPrototype**
- The ComponentPrototype references the SW-C Type
- If the SW-C Type supports **multiple instantiation**, the same type can be instantiated in many ComponentPrototypes
- **Example:** Door lock application
 - the same SW-C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



Per Instance Memory

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

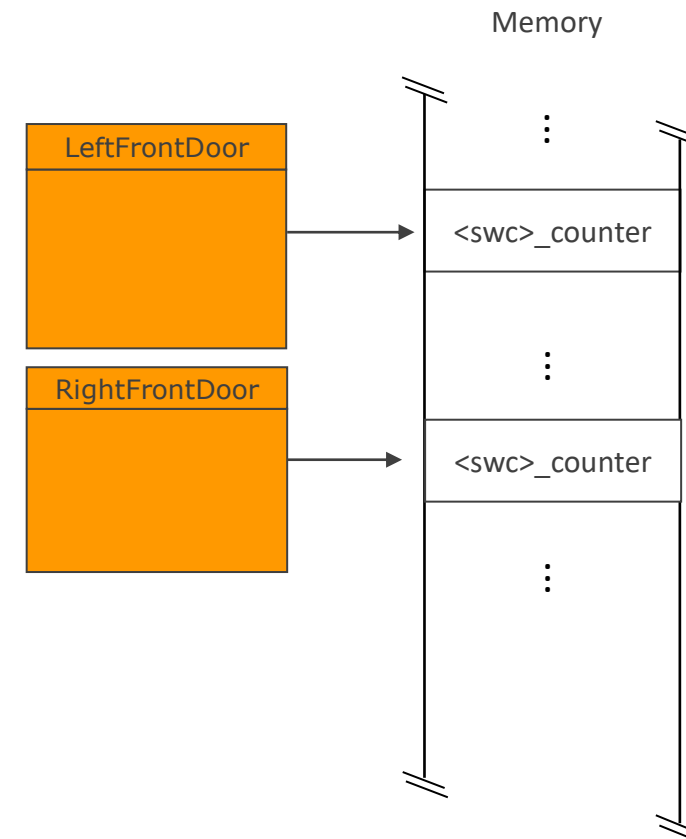
```
static uint8 counter;  
  
void myRunnable(Rte_Inst_SWC Rte_Instance)  
{  
    /* ... */  
    counter++;  
    /* ... */  
}
```



Per Instance Memory

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

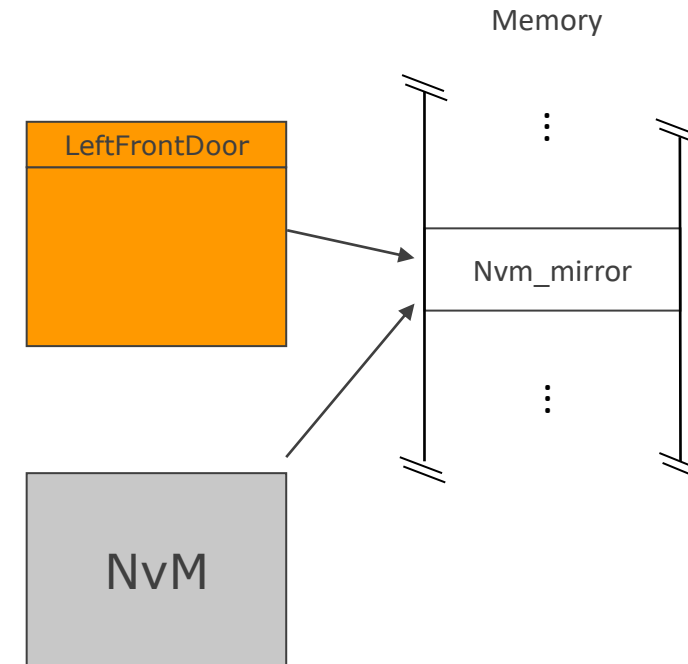
```
void myRunnable(Rte_Inst_SWC instance) {  
    /* ... */  
    cpnt = Rte_Pim_Counter(instance);  
    (*cpnt)++;  
    /* ... */  
}
```



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 SW-C concepts exclusive areas



Elektrobit



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*

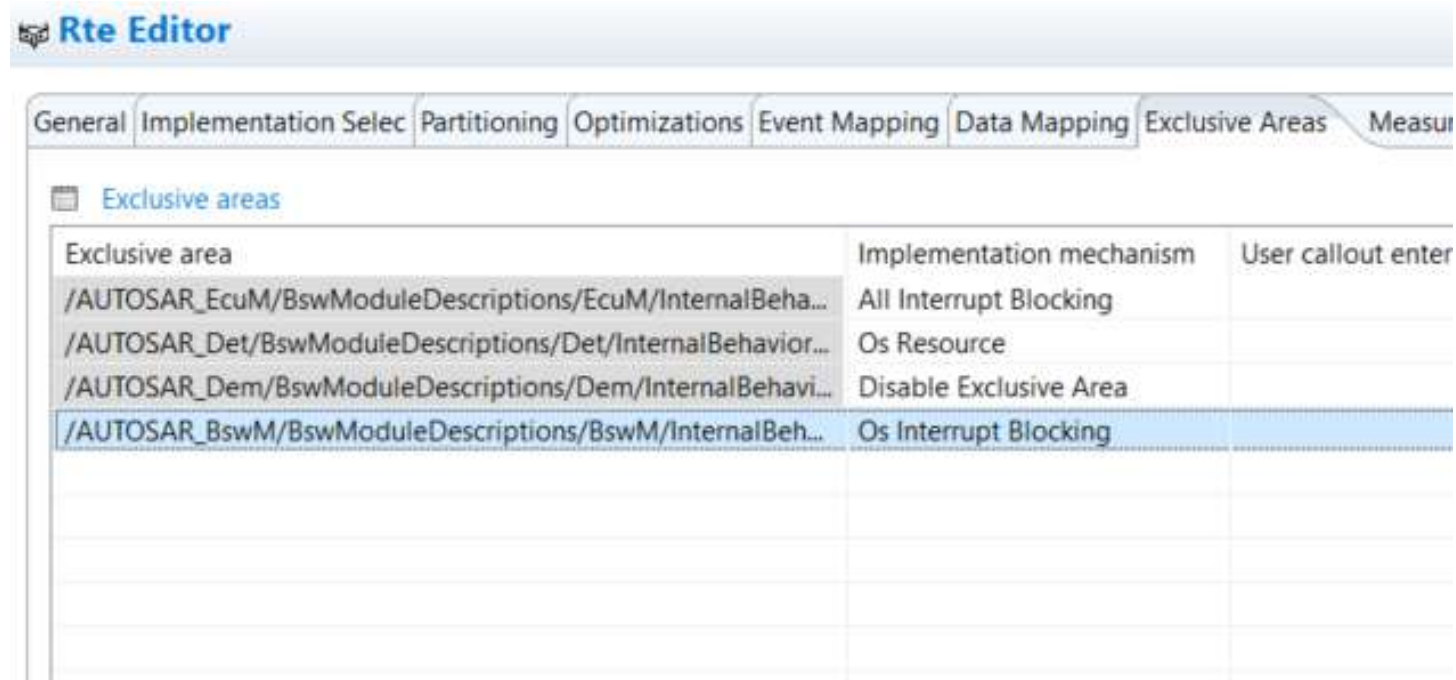
- An Exclusive Area can also be configured to protect an entire Runnable function
- To enable this feature, the attribute ***RunsInsideExclusiveArea*** is configured for the runnable:

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

- If using this approach, there is no need to use Rte_Enter()/Rte_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 SW-C concepts mode management



Elektrobit



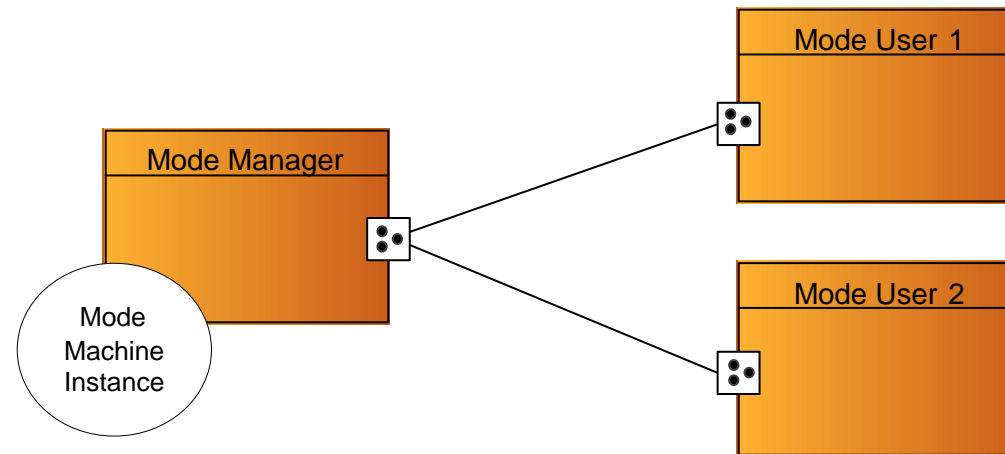
Mode Management

- **Modes** can be used to track system state changes
- Defining modes works as follows:
 - A **ModeDeclarationGroup** defines the different modes
 - In a **ModeSwitchInterface**, 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 acts as a **Mode Manager** – the only one allowed to make state changes via its PPort
- The **Mode Users** listen to mode changes by connecting their RPorts to the manager's PPort



- The RTE provides a **Mode Machine Instance** to the manager which takes care of all mode changes

MODE MANAGEMENT - *Mode manager APIs*

- The Mode Manager uses 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  
    }  
}
```

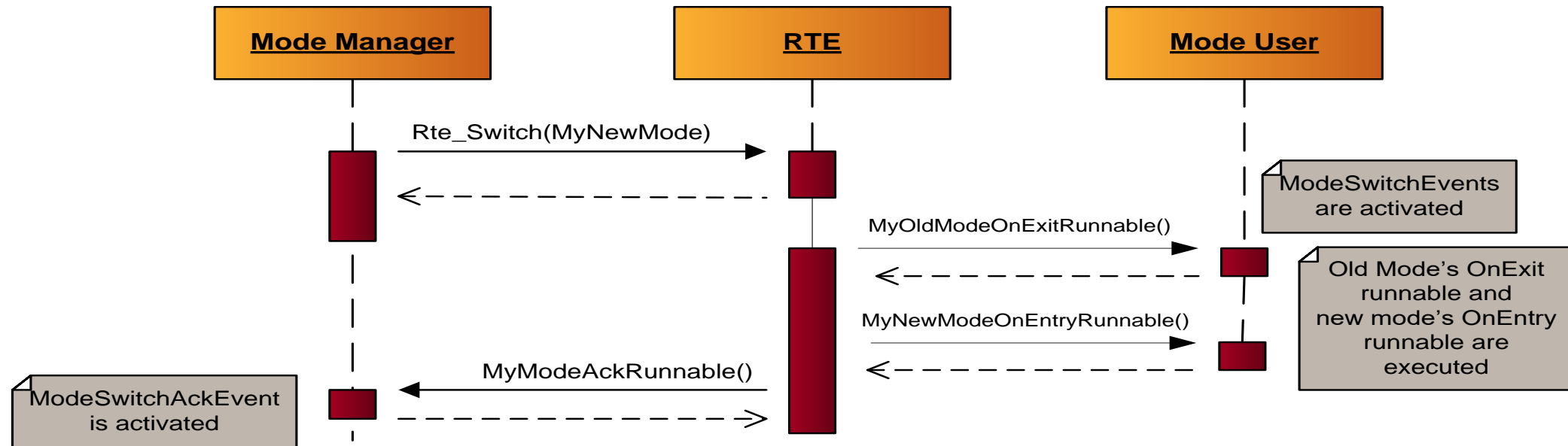
MODE MANAGEMENT - *Mode user APIs*

- The Mode Users 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*

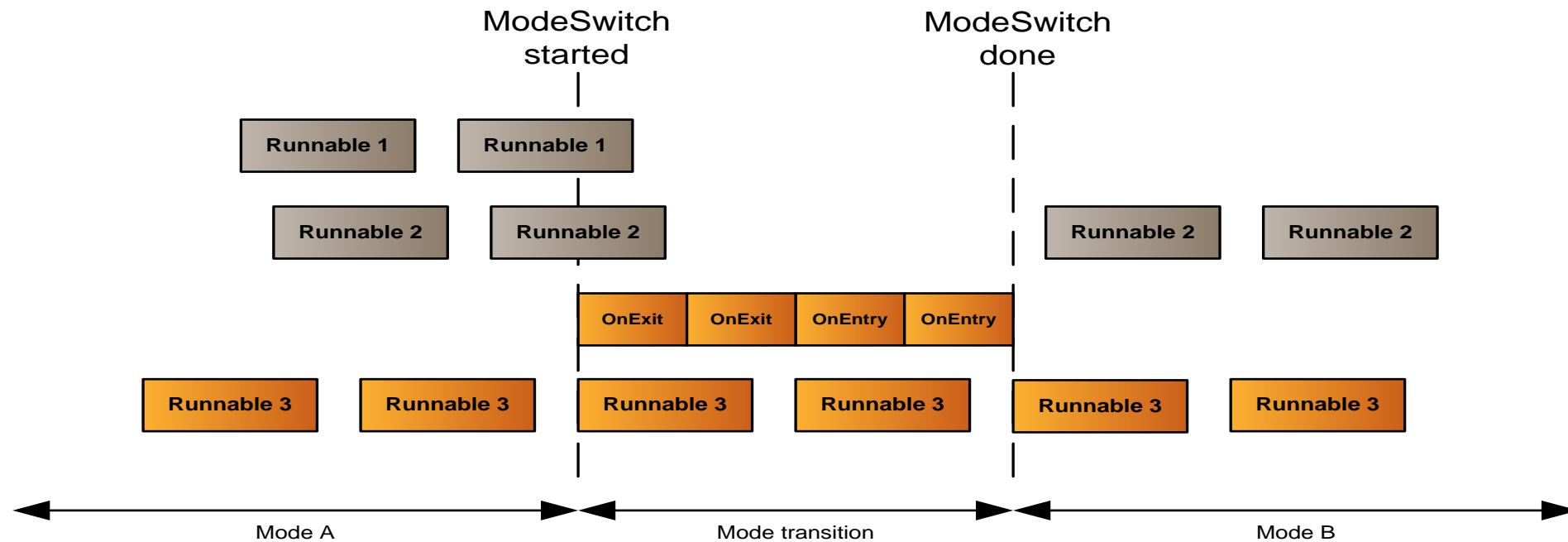
- A mode user can use **ModeSwitchEvents** to trigger **OnExit**, **OnTransition***, and **OnEntry** runnables upon mode change
- The Mode Manager can get an acknowledgement that all OnEnter/Exit runnables are finished via the **ModeSwitchAckEvent**



* Not illustrated

MODE MANAGEMENT - Mode disabling dependencies

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



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

Section Summary – Advanced SW-Cs concepts

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

Further Information



Elektrobit



RTE Documentation

More information on the EB tresos AutoCore Generic RTE can be found in the RTE documentation [AutoCore_Generic_RTE_documentation.pdf](#).

Among other information you can find there

- Module release notes including
 - Deviations and
 - Limitations with respect to the AUTOSAR standard
- A user's guide
- Module references including
 - Configuration parameter descriptions and
 - API descriptions



Elektrobit

EB tresos® AutoCore Generic 8 RTE documentation

product release 8.8.1



Get in touch!



Elektrobit

sales@elektrobit.com
www.elektrobit.com

