Communication type	RTEEvent →10.3	Chapter	unblock WP
Sender-Receiver (→7.5)	DataReceivedEvent	→10.3.1	(WP)
	DataReceiveErrorEvent	→10.3.2	
	DataSendCompletedEvent (explicit)	→10.3.3	(WP)
	DataWriteCompletedEvent (implicit)	→10.3.3	
Mode Switch (→7.9)	SwcModeSwitchEvent	→10.3.4	
	ModeSwitchedAckEvent	→10.3.5	(WP)
Client-Server (→7.4)	OperationInvokedEvent	→10.3.6	
	AsynchronousServerCallReturnsEvent	→10.3.7	(WP)
Other	TimingEvent	→10.3.8	
	BackgroundEvent	→10.3.9	
	ExternalTriggerOccurredEvent	→10.3.10	
	InternalTriggerOccurredEvent	→10.3.11	

RTE return codes are of type Std_ReturnType (→5.6.2).			
#define	hex	dec	Description
RTE_E_OK	0x00	0	No error.
RTE_E_INVALID	0x01	1	Returned by **Rte_Read** or **Rte_IStatus** signalizing invalid data element(s), dependent on the InvalidationPolicy →7.5.1.1
RTE_E_LOST_DATA	0x40	64	If new data is received and the queue is already full then the RTE discards the new data and sets an error flag. For the next read on the queue the **Rte_Receive** call returns the available data together with a status where the flag **RTE_E_LOST_DATA** is set →7.5.2.2.10 Note that **RTE_E_LOST_DATA** is an Overlayed Error →5.6.2.2
RTE_E_MAX_AGE_EXCEEDED	0x40	64	An **Rte_Read** or **Rte_IStatus** call indicates that the available data has exceeded the aliveTimeout limit. →7.5.2.2.4 Note that **RTE_E_MAX_AGE_EXCEEDED** is an Overlayed Error →5.6.2.2
RTE_E_COM_STOPPED	0x80	128	An IPDU group was disabled while the application was waiting for the transmission acknowledgment. No value is available. This is not necessarily considered a fault.
RTE_E_TIMEOUT	0x81	129	A blocking API call (**Rte_Receive**, **Rte_Call**, **Rte_SwitchAck**) returned due to expiry of a local timeout →8.4.4.3.1. OUT buffers are not modified.
RTE_E_LIMIT	0x82	130	An internal RTE limit (like queue size) has been exceeded (e.g. for **Rte_Send**, **Rte_Call**, **Rte_Switch**, **Rte_Trigger**, **Rte_IrTrigger**). OUT buffers are not modified. See example for (at least one) full queue →7.5.2.2.10
RTE_E_NO_DATA	0x83	131	No data was available for the API call. This is not (necessarily) to be considered as error. OUT buffers are not modified.
RTE_E_TRANSMIT_ACK	0x84	132	Transmission acknowledgement received. →7.5.2.1.3
RTE_E_NEVER_RECEIVED	0x85	133	No data received since system start or Partition restart. See parameter handleNeverReceived →7.5.2.2.7.
RTE_E_UNCONNECTED	0x86	134	The corresponding Port used for communication is not connected.→7.11
RTE_E_IN_EXCLUSIVE_AREA	0x87	135	The RunnableEntity (→8.4) could not enter a wait state because another RunnableEntity of the current Task (→10.5) call stack is running in an ExclusiveArea (→8.6).
RTE_E_SEG_FAULT	0x88	136	The parameters contain a direct or indirect reference to memory that is not accessible from the caller's Partition. →17

5.6.2.4 Predefined Error Codes

RTE return codes name (#define)	Value (hex, dec)	Description
RTE_E_OK	0x00	No error.
RTE_E_INVALID	0x01 1	Returned by **Rte_Read** or **Rte_IStatus** signalizing invalid data element(s), dependent on the `InvalidationPolicy` →7.5.1.1
RTE_E_LOST_DATA	0x40 64	If new data is received and the queue is already full, then the RTE discards the new data and sets an error flag. For the next read on the queue, the **Rte_Receive** call returns the available data together with a status where the flag **RTE_E_LOST_DATA** is set →7.5.2.2.10 Note that **RTE_E_LOST_DATA** is an Overlayed Error →5.6.2.2
RTE_E_MAX_AGE_EXCEEDED	0x40 64	An **Rte_Read** or **Rte_IStatus** call indicates that the available data has exceeded the `aliveTimeout` limit. →7.5.2.2.4 Note that **RTE_E_MAX_AGE_EXCEEDED** is an Overlayed Error →5.6.2.2
RTE_E_COM_STOPPED	0x80 128	An IPDU group was disabled while the application was waiting for the transmission acknowledgment. No value is available. This is not necessarily considered a fault.
RTE_E_TIMEOUT	0x81 129	A blocking API call (**Rte_Receive**, **Rte_Call**, **Rte_SwitchAck**) returned due to expiry of a local `timeout`. OUT buffers are not modified. See parameter `timeout` →8.4.4.3.1
RTE_E_LIMIT	0x82 130	An internal RTE limit (like queue size) has been exceeded (e.g. for **Rte_Send**, **Rte_Call**, **Rte_Switch**, **Rte_Trigger**, **Rte_IrTrigger**). OUT buffers are not modified. See example for (at least one) full queue →7.5.2.2.10
RTE_E_NO_DATA	0x83 131	No data was available for the API call. This is not (necessarily) to be considered as error. OUT buffers are not modified.
RTE_E_TRANSMIT_ACK	0x84 132	Transmission acknowledgement received. →7.5.2.1.3
RTE_E_NEVER_RECEIVED	0x85 133	No data received since system start or Partition restart. See parameter `handleNeverReceived` →7.5.2.2.7.
RTE_E_UNCONNECTED	0x86 134	The corresponding Port used for communication is not connected. →7.11
RTE_E_IN_EXCLUSIVE_AREA	0x87 135	The `RunnableEntity` (→8.4) could not enter a wait state because another `RunnableEntity` of the current Task (→10.5) call stack is running in an `ExclusiveArea` (→8.6).
RTE_E_SEG_FAULT	0x88 136	The parameters contain a direct or indirect reference to memory that is not accessible from the caller's Partition. →17

Table 5-38 Std_ReturnType

5.6.3 Standard Symbols

The Standard Symbols are provided by AUTOSAR and defined in file **Std_Types.h**.

Symbol	Value
`E_OK`	0x00
`E_NOT_OK`	0x01
`STD_HIGH`	0x01
`STD_LOW`	0x00

Symbol	Value
`STD_ACTIVE`	0x01
`STD_IDLE`	0x00
`STD_ON`	0x01
`STD_OFF`	0x00

```
          <HANDLE-OUT-OF-RANGE>EXTERNAL-REPLACEMENT</HANDLE-OUT-OF-RANGE>
          <USES-END-TO-END-PROTECTION>false</USES-END-TO-END-PROTECTION>
          <ALIVE-TIMEOUT>0.0</ALIVE-TIMEOUT>
          <ENABLE-UPDATE>false</ENABLE-UPDATE>
          <HANDLE-NEVER-RECEIVED>false</HANDLE-NEVER-RECEIVED>
          <HANDLE-TIMEOUT-TYPE>NONE</HANDLE-TIMEOUT-TYPE>
          <INIT-VALUE>
            <REFERENCE-VALUE-SPECIFICATION>
              <REFERENCE-VALUE-REF DEST="PARAMETER-DATA-PROTOTYPE">
                /ECU1/SwctExtRepExample/IbhSwctExtRepExample/DefaultSpeed
              </REFERENCE-VALUE-REF>
            </REFERENCE-VALUE-SPECIFICATION>
          </INIT-VALUE>
        </NONQUEUED-RECEIVER-COM-SPEC>
      </REQUIRED-COM-SPECS>
      <REQUIRED-INTERFACE-TREF DEST="SENDER-RECEIVER-INTERFACE">/ECU1/IfSR1</REQUIRED-INTERFACE-TREF>
    </R-PORT-PROTOTYPE>
  </PORTS>
  <INTERNAL-BEHAVIORS>
    <SWC-INTERNAL-BEHAVIOR>
      <SHORT-NAME>IbhSwctExtRepExample</SHORT-NAME>
      <CONSTANT-MEMORYS>
        <PARAMETER-DATA-PROTOTYPE>
          <SHORT-NAME>DefaultSpeed</SHORT-NAME>
          <TYPE-TREF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">/ECU1/DtSpd1</TYPE-TREF>
        </PARAMETER-DATA-PROTOTYPE>
      </CONSTANT-MEMORYS>
…
    </SWC-INTERNAL-BEHAVIOR>
  </INTERNAL-BEHAVIORS>
</APPLICATION-SW-COMPONENT-TYPE>
```

7.5.2.2.2 handleOutOfRangeStatus

The parameter `handleOutOfRangeStatus` defines how return values are created in case of an out-of-range situation:

- `silent`
- `indicate`

7.5.2.2.3 Maximum Delta Counter Init Value (maxDeltaCounterInit)

The `maxDeltaCounterInit` defines the initial maximum allowed gap between two counter values of two consecutively received valid data, i.e. how many subsequent lost data is accepted.

For example, if the receiver gets Data with counter 1 and MaxDeltaCounterInit is 1, then at the next reception the receiver can accept Counters with values 2 and 3, but not 4. Note that if the receiver does not receive new Data at a consecutive read, then the receiver increments the tolerance by 1.

See also →18.4 E2E Initial Maximum Gap (maxDeltaCounterInit)

7.5.2.2.4 Alive Timeout (aliveTimeout)

The mandatory parameter `aliveTimeout` specifies the amount of time (in seconds) after which the reception of data “times out” while the data element has not been received. The monitoring functionality is provided by the COM module. The RTE transports the event of reception timeouts to Software Components as “data element outdated”. For such an event a `RunnableEntity` can be invoked via a `DataReceiveErrorEvent` (→10.3.2). →7.5.2.2.8

Additionally the **Rte_Read** or **Rte_IStatus** API calls will have the flag **RTE_E_MAX_AGE_EXCEEDED** set in their return value.

If the `aliveTimeout` attribute is set to 0 (zero) then no timeout monitoring will be performed. This parameter is not applicable for communication that is local (within a Partition → 17.2). If `aliveTimeout` is present and the communication is between different Partitions of the same ECU, time-out monitoring is disabled. Instead, a

7.5.2.2.8 Timeout Type (handleTimeoutType)

The mandatory attribute handleTimeoutType controls the behavior regarding receive-timeouts (aliveTimeout →7.5.2.2.4):

- none: receive-value will not be replaced.
- replace: receive-value will be replaced by the value in ComInitValue.

7.5.2.2.9 Non-Queued S/R-Communication

If swImplPolicy (→5.15) of a VariableDataPrototype (→5.9.2) is not set to queued then last-is-best semantics applies. Every newly received data overwrites already existing data whether the existing data was read by the application or not. Implicit or explicit read access always returns the last received data.

7.5.2.2.10 Queued S/R-Communication (queueLength)

In the ComSpec of the Port the elements of the corresponding Port Interface can be defined as queued or non-queued. The swImplPolicy (→5.15) of a VariableDataPrototype (→5.9.2) indicates the way how it shall be processed at the receiver's side. If set to queued the VariableDataPrototype needs to be added to a queue from which it is (later) consumed by the actual receiver Software Component (the queue is first-in-first-out - FIFO). If swImplPolicy is set to any other valid value, then the last-is-best semantics applies. →7.5.2.2.9

Please note that the swImplPolicy (→5.15) of the referenced dataElement needs to be compatible with the Port ComSpec. So for a QueuedSenderComSpec or QueuedReceiverComSpec the swImplPolicy of the dataElement must be configured as queued.

If new data is received and the queue is already full, then the RTE discards the new data and sets an error flag while the send-API returns **RTE_E_LIMIT**. For the next read on the queue the RTE returns the available data together with a status where the flag **RTE_E_LOST_DATA** is set.

When reading an empty queue, the RTE is returning status **RTE_E_NO_DATA**. In this case the returned data is undefined.

Example:

For communication between two ApplicationSwComponentTypes **SwctAppSR1** and **SwctAppSR2** we have the S/R Port Interface **IfSendId** defined with data element **SendId** of ApplicationDataType **DtSendId**.

```
/* Receiver Runnable invoked by On-Receive-Event EvDr1 */
void Receiver( void )
{
  DtImplSendId NewData = 0;
  Std_ReturnType status = RTE_E_OK;

  do
  {
    status = Rte_Receive_RpReceiverPort_SendId( &NewData );

    if( RTE_E_LOST_DATA == (status & RTE_E_LOST_DATA) )
    {
      /* Queue overflow (lost data) flag is set */
      /* Do whatever is necessary… */

      /* Remove overflow (lost data) flag for further processing */
      status &= ~RTE_E_LOST_DATA;
    }

    if(RTE_E_NO_DATA == status )
    {
      /* RTE_E_NO_DATA indicates that the queue is empty.
       * Therefore exit the loop */
      break;
    }
    else
    {
      /* do whatever necessary with NewData */
    }
  } while( RTE_E_OK == status );
}
```

Note that the queue-emptying on the receiver side is not handled by the RTE and that only the data is queued, not the events (**EvDr1**)! It is therefore necessary for the Receiver-Runnable to empty the queue, as illustrated in the example. Otherwise, the queue will eventually overflow.
If the sender and the receiver Runnables are assigned to the same Task (→10.5), then the **ReReceiver** is only called once after the sender filled the queue. That means that **ReReceiver** will only be invoked once and the code has to empty the "receiver queue" in a loop as demonstrated in the example above.
If the sender and the receiver Runnables are running on different Tasks then the **ReReceiver** could start emptying the queue while the sender is still filling the queue. In this case there might be more than one `DataReceivedEvent` per "Sender-burst".

7.5.2.2.10.1 Queued with n:1 Communication

On a queued communication with n:1 connections, one queue is implemented by the RTE on the receiver side as illustrated in the figure below and in the following example.

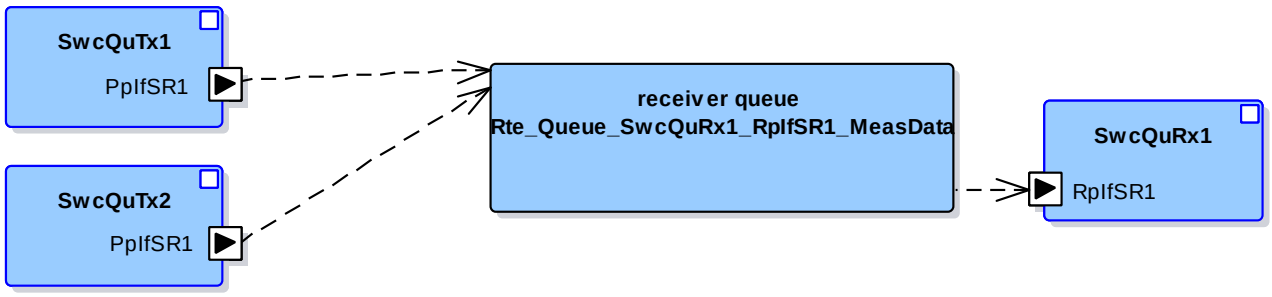

Figure 7-4 Queued S/R communication n:1

Code example of the generated RTE code writing and reading to and from the queue: Sender components are of the same `SwComponentType` and named **SwcQuTx1** and **SwcQuTx2**. The Send functions both fill the same queue **Rte_Queue_SwcQuRx1_RpIfSR1_MeasData** for the receiver component **SwcQuRx1**:

```
FUNC(void, QuRxType_CODE) ReQuRxRun1( P2CONST(struct Rte_CDS_QuRxType, AUTOMATIC, RTE_APPL_CONST) self )
{
  uint32 data = 0U;
  Std_ReturnType retValue = RTE_E_OK;

  /* Empty queue in a loop (maybe there is more than one element in the queue) */
  do
  {
    retValue = Rte_Receive_RpIfSR1_MeasData( inst, &data );

    if( RTE_E_LOST_DATA == (retValue & RTE_E_LOST_DATA) )
    {
      /* Queue overflow (lost data) flag is set */
      /* Do whatever is necessary… */

      /* Remove overflow (lost data) flag for further processing */
      retValue &= ~RTE_E_LOST_DATA;
    }

    switch( retValue )
    {
      case RTE_E_NO_DATA:
        /* queue is empty */
        break;
      case RTE_E_TIMEOUT:
        /* no data received – timeout occurred */
        break;
      case RTE_E_OK:
        /* everything is fine */
        break;
      default:
        /* something went wrong*/
        break;
    }

  /* RTE_E_NO_DATA indicates that the queue is empty. So, loop should be exited */
  } while( RTE_E_OK == retValue );
}
```

Generated code for RTE where the data to be sent is added to the RTE queue **Rte_Queue_SwctQuRx1_RpIfSR1_MeasData**:

```
#define Rte_Send_PpIfSR1_MeasData(inst, data) ((inst)->PpIfSR1.Send_MeasData(data))
#define Rte_Feedback_PpIfSR1_MeasData(inst) ((inst)->PpIfSR1.Feedback_MeasData())

static FUNC(Std_ReturnType, RTE_CODE) Rte_Send_SwctQuTx1_PpIfSR1_MeasData( uint32 data )
{
  Std_ReturnType retVal = RTE_E_OK;
  Rte_FBStatus_SwctQuTx1_PpIfSR1_MeasData = RTE_E_NO_DATA;

  SuspendOSInterrupts();
  if(FALSE != Rte_Queue_Full(Rte_Queue_SwctQuRx1_RpIfSR1_MeasData))
  {
    Rte_Queue_Set_Overflow(Rte_Queue_SwctQuRx1_RpIfSR1_MeasData);
    retVal = RTE_E_LIMIT;
  }
  else
  {
    Rte_Queue_Push(Rte_Queue_SwctQuRx1_RpIfSR1_MeasData, data);
  }
  ResumeOSInterrupts();

  Rte_FBStatus_SwctQuTx1_PpIfSR1_MeasData = RTE_E_TRANSMIT_ACK;
  return retVal;
}

static FUNC(Std_ReturnType, RTE_CODE) Rte_Feedback_SwctQuTx1_PpIfSR1_MeasData( void )
{
  return Rte_FBStatus_SwctQuTx1_PpIfSR1_MeasData;
}

#define Rte_Receive_RpIfSR1_MeasData(inst, data) ((inst)->RpIfSR1.Receive_MeasData(data))
```