Software Component Language

Syntax Reference

Naming

To name an element matching one of the keywords in ARText, please use the "^" character before the name. For example, to name an integer element "int", which matches the reserved keyword "int" in ARText, specify the element as such:

```
int ^int
```

The AUTOSAR model generated will not include this special "^" character if it is specified in ARText.

Package

Each ARText file must specify a package. All the elements that are specified in an ARText file will be created inside the specified package. A package is declared by the keyword **package** followed by the package name at the beginning of an ARText file:

```
package RootPackage
...
```

Sub-packages can be specified by the complete package path where each package is separated by '.' similar to package declarations in Java. The following examples results in the creation of a package "SubPackage" inside of a package "RootPackage".

```
package RootPackage.SubPackage
...
```

Import statements

Import statements are listed after the package declaration.
This feature allows users to import AUTOSAR elements from different packages, enabling users to reference these imported elements with only the short name rather than the fully qualified name.

There are two types of import statements, the first imports the specific AUTOSAR element, the second uses the wild card character "*" to indicate that all elements from the AUTOSAR element are imported.

For example:

```
package components
import interfaces.srInterface
import interfaces.csInterface
import interfaces.paramInterface
import datatypes.*
...
```

Units

A unit declaration consists of:

- the **unitDeclaration** keyword
- the short name of the unit
- optional **factorSiToUnit** keyword followed by its double value

- optional **offsetSiToUnit** keyword followed by its double value
- optional reference to a PhysicalDimension. See below on how to declare these.

For example:

```
package myPackage
unitDeclaration unit1
unitDeclaration unit2 factorSiToUnit 1.0
unitDeclaration unit3 offsetSiToUnit 2.0
unitDeclaration unit4 aPhysicalDimension
unitDeclaration unit5 factorSiToUnit 3.0 offsetSiToUnit 4.0 aPhysicalDimension

physcialDimension aPhysicalDimension
```

Physical Dimensions

Physical dimensions can be referenced by Units.

The physical dimension declaration consists of:

- the **physicalDimension** keyword
- the short name of the physicalDimension
- optional **currentExp** keyword followed by its decimal value
- optional **lengthExp** keyword followed by its decimal value
- optional **luminousIntensityExp** keyword followed by its decimal value
- optional **massExp** keyword followed by its decimal value
- optional **molarAmountExp** keyword followed by its decimal value
- optional **temperatureExp** keyword followed by its decimal value
- optional **timeExp** keyword followed by its decimal value.

For example:

```
package myPackage

physcialDimension physicalDimension0

physcialDimension physicalDimension1 currentExp 1.0

physcialDimension physicalDimension2 currentExp 1.0
                              lengthExp 2.0
                              luminousIntensityExp 3.0
                              massExp 4.0
                              molarAmountExp 5.0
                              temperatureExp 6.0
                              timeExp 7.0

physcialDimension physicalDimension3 lengthExp 2.0
                              massExp 4.0
                              temperatureExp 6.0
```

Data types

The following sections describe the available primitive and composite data types available within ARText.

Boolean

The boolean declaration consists of:

- the **boolean** keyword
- the short name of the data type
- optional invalid value keyword **invalidValue** followed by the value (**true** or **false**)
- optional **extends** keyword followed by the reference to a software base data type, or an implementation data type (AUTOSAR 4x).

For example:

```
boolean myBoolean
boolean myBoolean2 invalidValue false
boolean myBoolean3 extends BooleanBaseType
```

Integer

An integer declaration consists of:

- the **int** keyword
- the short name of the data type
- optional **min** keyword to specify the lower limit
    - followed by the lower limit value (of type long)
    - optional: the interval type, which can be either: **closed**, **open** , **infinite**. The default type is **closed**.
- optional **max** keyword to specify the upper limit
    - followed by the upper limit value (of type long)
    - optional: the interval type, which can be either: **closed**, **open** , **infinite**. The default type is **closed**.
- optional **unit** keyword followed by the name of a referenced unit declaration
- optional data constraints association. Use the **constraint** keyword, followed by a reference to the data constraint.
- optional invalid value keyword **invalidValue** followed by an integer value.
- optional **extends** keyword followed by the reference to a software base data type, or an implementation data type (AUTOSAR 4x).

For example:

```
int myInt1
int myInt2 min 8 open max 9 closed
int myInt3 max 9
int myInt4 min 8
int myInt5 invalidValue -1
int myInt5 extends IntegerBaseType
int myInt min 1 max 5 constraint myConstraint
```

Please note:
If minimum and maximum limits are supplied, the default interval type will be **CLOSED**.

AUTOSAR 4.0
In this release, minimum and maximum limits are specified as data constraint rules. Therefore both the **min**, **max** and **constraint** keywords cannot be used together. If you require both min, max and extra data constraint rules, please create a data constraint containing all these rules and add a reference in the integer type declaration. However if you just require the minimum and maximum limits, and no other additional data constraints, you may use the **min** and **max** keywords.

Real

A real declaration consists of:

- the **real** keyword
- the short name of the data type
- optional **min** keyword to specify the lower limit
    - followed by the lower limit value (of type double)
    - optional: the interval type, which can be either: **closed**, **open** , **infinite**. The default type is **closed**.
- optional **max** keyword to specify the upper limit
    - followed by the upper limit value (of type double)
    - optional: the interval type, which can be either: **closed**, **open** , **infinite**. The default type is **closed**.
- optional **encoding** keyword followed by either **double** or **single**
- optional **allowNaN** keyword, specifying this keyword indicates that the real type allows NaN, without the keyword, the real type does not allow NaNs

the real type does not allow NaNs.

- optional **unit** keyword followed by the name of a referenced unit declaration
- optional invalid value keyword **invalidValue** followed by a real value.
- optional **extends** keyword followed by the reference to a software base data type, or an implementation data type (AUTOSAR 4x).
- optional **constraints** keyword followed by a reference to a data constraint.

> **Please note:**
> The min, max and invalidValue values are of type double, however the following special values are also valid: **NaN** (not a number), **INF** (positive infinity) and **-INF** (negative infinity).

For example:

```
real myRealSingle
real myRealSingle2 max 0.1 open
real myRealSingle3 min 8.9
real myReal encoding double
real myReal allowNaN
real myReal invalidValue 0.0
real myReal extends refSwRealBaseType constraint myConstraint
```

String

A string declaration consists of:

- the **string** keyword
- the short name of the string
- the **length** keyword followed by the maximum number of characters for this string
- the **encoding** keyword followed by the encoding of the string in quotes
- optional invalid value keyword **invalidValue** followed by a string value
- optional **extends** keyword followed by the reference to a software base data type, or an implementation data type (AUTOSAR 4x).

For example:

```
string myString length 8 encoding "ISO-8859-1"
string myString2 length 9 encoding "ISO-8859-1" invalidValue "NULL"
```

Software Base Types

All primitive types may be associated with a *software base type* (Primitive types include: Boolean, Integer, Real, and String).

To specify a software base type, after declaring the primitive type, specify the **extends** keyword followed by a reference to the software base type. Note that ARText does not provide the syntax to define software base types, however these types can be referenced from arxml files.

Specifying a software base type is optional for all primitive types.

For example:

```
int myInt min 1 max 5
        extends UInt8
```

Data Constraints

A data constraint consists of:

- the **dataConstraint** keyword
- the data constraint short name
- enclosed within braces, are optional physical and internal constraints, these consist of:

- the **rule** keyword
- the **physical** or **internal** keyword
- the **min** keyword followed by the minimum range value
- the **max** keyword followed by the maximum range value.
- (for physical constraints from AUTOSAR 3.x and up) an optional **unit** keyword followed by a reference to a Unit.

For example:

```
dataConstraint myConstraint {
        rule physical min 3 max 38 unit Units.meter
        rule internal min 8 max 19
}
```

> **AUTOSAR 4.x**
> In AUTOSAR 4.x, the minimum and maximum limits are now defined as data constraints. Therefore, if the integer type has a reference to a specified data constraint, the minimum declared for the type must be the lowest physical limit out of all data constraint rules, and similarly, the maximum declared for the type must be the uppermost physical limit out of all the data constraints.
>
> If the referenced data constraint has no physical data constraint declared, then one will be created if the integer "min" and "max" values have been supplied.

Array

An array declaration consists of:

- the **array** keyword
- the reference to the data type that this array will hold elements from
- in square brackets the length of the array
- the short name of the array data type.

For example:

```
array myBoolean[8] myArray
```

Record

A record declaration consists of:

- the **record** keyword
- the short name of the record data type
- a list of comma separated record elements specified within braces. A record element consists of:
  - a reference to an existing data type
  - and the short name of the record element.

For example:

```
record myRecord {
        myInt myElement1,
        myBoolean myElement2,
        myString myElement3
}
```

Enumerations

Enumerations in AUTOSAR are not primitive datatypes. Rather a range of integers can be used as a structural description. The mapping of the integers on "labels" in the enumeration is seen as Data-Semantics and not as part of the structural description. In AUTOSAR, CompuMethods are used for the conversion of internal values into their physical representation and vice versa. ARText simplifies the definition of enumerations by providing a specific enumeration command. A enumeration is defined by:

- the **enum** keyword
- the shortName of the Enumeration
- an optional lower limit after the keyword **min**
- an optional upper limit after the keyword **max**
- within braces: a comma separated list of enumeration literals. An literal is defined by an identifier and an optional value. If no value is provided, the values for the enumerations will be automatically generated. The values can either be integer or hexa-decimal. However, all enumeration values must be specified consistently, i.e. all without values, or all with values.

For example:

```
enum booleanEnum {TRUE, FALSE}
enum intStatus {OK = 1 , ERROR = -1}
enum hexStatus {OK = 0x01 , ERROR = 0x02}
enum hexStatus {OK = 0x01 , ERROR = 0x02}
enum myBoolean min 0 max 1 {
                TRUE = 0, FALSE = 1
}
```

As an example in AUTOSAR 2.1, this enumeration is translated into the following CompuMethod:

```
<COMPU-METHOD>
    <SHORT-NAME>booleanEnum</SHORT-NAME>
    <CATEGORY>TEXTTABLE</CATEGORY>
    <COMPU-INTERNAL-TO-PHYS>
        <COMPU-SCALES>
            <COMPU-SCALE>
            <LOWER-LIMIT INTEVAL-TYPE="CLOSED">0</LOWER-LIMIT>
            <UPPER-LIMIT INTEVAL-TYPE="CLOSED">0</UPPER-LIMIT>
            <COMPU-CONST>
                <VT>false</VT>
            </COMPU-CONST>
            </COMPU-SCALE>
            <COMPU-SCALE>
                <LOWER-LIMIT INTEVAL-TYPE="CLOSED">1</LOWER-LIMIT>
                <UPPER-LIMIT INTEVAL-TYPE="CLOSED">1</UPPER-LIMIT>
                <COMPU-CONST>
                    <VT>true</VT>
                </COMPU-CONST>
            </COMPU-SCALE>
        </COMPU-SCALES>
    </COMPU-INTERNAL-TO-PHYS>
</COMPU-METHOD>
```

Fixed-point datatypes

A value of a fixed-point data type is an integer that is scaled by a specific factor **m** with an additional bias **b**.

```
F = m * x + b
```

An fixed-point datatype declaration consists of:

- the **fixed** keyword
- the short name of the data type
- the fractional slope (as a double) of the data type after the **slope** keyword.
- an optional bias (as a double) of the scaling of the data type after the optional **bias** keyword.
- additional parameters similar to the integer data type

```
fixed myFixed1 slope 1.1 bias 1.0
fixed myFixed2 slope 1.1 // bias defaults to 0
fixed myFixed3 slope 1.1 min 1 max 10 // for additional parameters see integer type
```

AUTOSAR 4.* specific Features

Application Data Types

Application datatypes can be defined by adding the flag **app** after the datatype's keyword:

```
enum app myBool {TRUE, FALSE}
int app myInt min 0 max 1
string app myString length 10 encoding "UTF8"
real app myReal min 0.0 max 1.0
array app myInt[0] myIntArray
record app myRecord{
        myInt field1,
        myInt field2
}
```

Implementation Data Types

Implementation datatypes can be defined by adding the flag **impl** after the datatype's keyword:

```
enum impl myBoolImpl {TRUE, FALSE}
int impl myIntImpl min 0 max 1
string impl myStringImpl length 10 encoding "UTF8"
real impl myReal min 0.0 max 1.0
array impl myIntImpl[0] myIntArray
record impl myRecord{
        myIntImpl field1,
        myIntImpl field2
}
```

Data Type Mapping Set

A data type mapping set is used to map application datatypes to implementation datatypes.

A data type mapping set declaration consists of:

- the **dataTypeMappingSet** keyword
- the name of the data type mapping set
- a list of mappings. Each mapping starts with the keyword **map** followed by a reference to the implementation datatype and then a reference to the application datatype or mode declaration group.

```
dataTypeMappingSet myMappingSet {
  map myImplementationType1 myApplicationType1
  map myImplementationType2 myApplicationType2
  map myImplementationType2 myApplicationType3
  map myImplementationType3 myModeDeclarationGroup
}
```

Constants

A constant declaration consists of:

- the **const** keyword
- a reference to the data type
- the short name of the constant
- the initial value

Constants also support nested arrays and record types.

Primitive type constants:

```
boolean MyBoolean
const MyBoolean MY_BOOLEAN = true
```

```
int MyInt
const MyInt MY_INT = -1

real MyReal
const MyReal MY_REAL = 1.0

enum MyEnum {TRUE, FALSE}
const MyEnum TRUE = :TRUE
```

Array type constants:

```
boolean MyBoolean
array MyBoolean[2] MyArray
const MyArray MyArrayConst = MyArray{
        true, false
}
```

If you would like to initialize an array with a default value rather than explicitly specifying the same value "x" times, you may use the **initAll** keyword.

```
boolean MyBoolean
array MyBoolean[200] MyArray
const MyArray MyArrayConst = MyArray initAll true
```

To initialize an array of a composite type, you may use a constant reference:

```
boolean myBoolean
array myBoolean[2] myArray
array myArray[10] myArrayArray
const myArray myConstant = myArray initAll true

// initialize the array with a constant reference
const myArrayArray myConstantArray = myArrayArray initAll myConstant
```

Or to initialize an array a composite value specification:

```
boolean myBoolean
array myBoolean[3] myArray
array myArray[9] myArrayArray
const myArray myConstant = myArray initAll true

// initialize the array with an composite value specification
const myArrayArray myConstantArray = myArrayArray
        initAll myArray { true, false }
```

Record type constants:

```
int MyInt
boolean MyBoolean
array MyBoolean[0] MyArray

record MyRecord {
        MyBoolean MyBooleanField,
        MyInt MyIntField
}

record MyNestedRecord {
        MyBoolean MyBooleanField,
        MyRecord MyRecordField
}

record MyArrayInRecord {
        MyBoolean MyBooleanField,
        MyRecordArray MyRecordArrayField
}
```

```
}

const MyRecord MY_RECORD = MyRecord{
        MyBooleanField = true,
        MyIntField = 0
}

const MyNestedRecord MY_NESTED_RECORD = MyNestedRecord {
        MyBooleanField = true,
        MyRecordField = MyRecord {
                MyBooleanField = true,
                MyIntField = 0
        }
}

const MyArrayInRecord MY_RECORD_WITH_ARRAY = MyArrayInRecord{
        MyBooleanField = true,
        MyRecordArrayField = MyRecordArray{
                MyRecord{
                        MyBooleanField = true,
                        MyIntField = 0
                },
                MyRecord{
                        MyBooleanField = true,
                        MyIntField = 0
                }
        }
}
```

Port Interfaces

Port interfaces declare services or data elements that are required and provided by the respective ports. Interface declarations start with the keyword **interface** followed by a keyword that indicates the interface type.

Sender Receiver Interface

Sender receiver interfaces allow for the specification of the typical asynchronous communication pattern where a sender provides data that is required by one or more receivers. While the actual communication takes place via the respective ports, sender receiver interfaces permit a formal description of the kind of data that is sent and received.
A sender receiver interface must contain at least one data element or mode group

The interface declaration consists of:

- the **interface** keyword
- the **senderReceiver** keyword
- an optional **service** parameter, this indicates that this is a service port interface which can be used for AUTOSAR services.
- the short name of the interface
- a list of data element declarations enclosed within braces

A data element consists of:

- the **data** keyword
- a reference to an existing data type
- the short name of the data element
- an optional **queued** keyword, specifying this indicates that the data element content is queued. Queued indicates that the data element has "event" semantics, i.e. data elements are stored in a queue and all data elements are processed in "first in first out" order. If it is not queued, then the "last in first out" semantics applies. Please note: Depending on the read access cycle to the data element some values might not be processed by the receiver.
- an optional **initValue** keyword, followed by a value that will match the data type specified for this data element.

For example:

```
interface senderReceiver service mySRInterface {
        // data elements
        data myBoolean myElement1 queued
        data myInt myElement2 initValue 23
        data myString myElement3
}
```

Client Server Interface

A client server interface declares a number of operations that can be invoked on a server by a client. They are the service oriented counterpart to the sender receiver interfaces.

A client server interface declaration consists of:

- the **interface** keyword
- the **clientServer** keyword
- an optional **service** parameter, this indicates that this is a service port interface which can be used for AUTOSAR services.
- the client server interface short name
- an optional list of the possible application errors, this is defined by:
    - the **error** keyword
    - the application error short name
    - the application error integer value
- a optional list of operation prototypes enclosed within braces, an operation prototype is defined by:
    - the **operation** keyword
    - the short name of the operation
    - operation declaration of the possible application errors, this is defined by:
        - the **possibleErrors** keyword
        - a comma separated list of application error references
    - within braces a list of arguments. Each argument is defined by:
        - a direction being one of **in inout out**, this specifies the direction that the arguments are passed between the client and server
        - a reference to an existing data type
        - the short name of the argument
        - (only supported in AUTOSAR 4.x) an optional declaration of the Server Argument Policy:
            - the **policy** keyword
            - the policy value being one of **useArgumentType useArrayBaseType useVoid**.

For example:

```
interface clientServer service myCSInterface {
        error myErrorA 1
        error myErrorB 2
        error myErrorC 3

        operation myOperation possibleErrors myErrorA, myErrorB {
                in myBoolean myArg1
                inout myInt myArg2
                out myString myArg3 policy useArgumentType
        }

        operation myOperation2{in myString myArg1}
}
```

Parameter Interface

A parameter interface declaration consists of:

- the **interface** keyword
- the **param** keyword
- an optional keyword **service**, if present indicates a service interface
- the parameter interface short name
- within braces, a list of parameter elements, each consisting of:
    - the **param** keyword
    - a reference to a data type

    - a reference to a data type
    - the short name of the data element
    - an optional category
    - an optional **initValue** keyword, followed by a value that will match the data type specified for this param element.

For example:

```
interface param service myParamInterface {
        param myBoolean myElement "value" initValue true
}
```

Mode Switch Interface

A mode switch interface declares a ModeDeclarationGroupPrototype to be sent and received. A mode switch interface declaration consists of:

- the **interface** keyword
- the **modeSwitch** keyword
- an optional keyword **service**, if present indicates a service interface
- the mode switch interface short name
- within braces, a list of mode group prototypes:

A mode group prototype describes the collection of mode switches that can be communicated via the interface. To declare these specify:

- the **mode** keyword
- a reference to an existing mode declaration group
- the mode declaration group prototype short name.

> Mode Switch Interfaces are only supported from AUTOSAR release 4.0. For previous AUTOSAR releases, a corresponding Sender Receiver Interface will be created.

Software Components

Software component declarations start with the keyword **component** followed by a keyword that indicates the type of the software component. Currently the following component types are supported:

1. Application Software Components
2. Service Software Components
3. Sensor Actuator Software Components
4. Parameter Software Components

Application Software Components

An application software component (former atomic software component) declaration consists of:

- the **application** keyword
- the software component short name
- within braces (each definition will be explained in more detail in the following sections):
    - a list of ports

For example:

```
component application myComponent {
        ports {
                /* ... */
        }

}
```

Service Software Components

An service software component declaration consists of:

- the **service** keyword
- the software component short name
- within braces (each definition will be explained in more detail in the following sections):
    - a list of ports
    - the internal behavior declaration
    - the component implementation

Parameter Software Components (former CalPrm Components)

An calibration parameter software component declaration consists of:

- the **param** keyword
- the software component short name
- within braces a list of param ports:
    - param ports are PPorts that can only provide an parameter interface (former calprm interface)
- within braces a list of data type mapping sets:
    - the **dataTypeMappings** keyword
    - a list of references to data type mapping sets.
    - This configuration option is only available for AUTOSAR 4.0 models.

```
component param mySensorActuatorComponentName {

        ports{
                param myParamPort provides aParamInterface
                param myParamPort provides aParamInterface
        }

        dataTypeMappings {
                myDataTypeMappingSet1
                myDataTypeMappingSet2
        }


}
```

Sensor Actuator Software Components

The SensorActuatorSoftwareComponentType is designed as a specialization of an application software component with an additional reference to a SensorActuatorHW.

SensorActuatorSoftwareComponentType needs to be mapped and run on exactly that ECU that contains the SensorActuatorHW that it refers to. And in contrast to an ApplicationSoftwareComponentType, an SensorActuatorSoftwareComponentType may use the I/O hardware abstraction directly.

The SensorActuatorSoftwareComponentType introduces the possibility to link from the software representation of a sensor/actuator to its hardware description provided by the ECU Resource Template.

An service software component declaration consists of:

- the **sensorActuator** keyword
- the software component short name
- within braces (each definition will be explained in more detail in the following sections):
    - a list of ports
    - the internal behavior declaration
    - the component implementation
    - a sensor and actuator hardware declaration.
- the **hw** keyword
- the reference to the sensor actuator hardware component

```
component sensorActuator mySensorActuatorComponentName {
        /* ports */
        hw refSensorHW
}
```

Ecu Abstraction Components

The EcuAbstractionSoftwareComponentType is the interface between sensor-actuator components and the IO hardware abstraction. An ecu abstraction component is specialized for a specific configuration of IO hardware abstraction for a specific ECU.

An ecu abstraction component type declaration consists of:

- the **ecuAbstraction** keyword
- the component type short name
- within braces (each definition will be explained in more detail in the following sections):
    - a list of ports
    - the internal behavior declaration
    - the component implementation

```
component ecuAbstraction myEcuAbstractionComponentType {
        ports {
                /* ... */
        }
}
```

Complex Device Driver Components

The ComplexDeviceDriverComponentType is used to model a function outside of the normal Autosar BSW software stack. They represent the interface between software components and the specialized software implementing the complex device driver.

An ecu abstraction component type declaration consists of:

- the **complexDeviceDriver** keyword
- the component type short name
- within braces (each definition will be explained in more detail in the following sections):
    - a list of ports
    - the internal behavior declaration
    - the component implementation

```
component complexDeviceDriver myComplexDeviceDriverComponentType {
        ports {
                /* ... */
        }
}
```

Ports

Ports are defined interaction points to describe the possible kinds of (data as well as service oriented) communication with other software components.
Ports are either require or provide ports. A require-port (short: r-port) requires certain services or data, while a provide-port (p-port) on the other hand provides those services or data.

The declaration of ports consists of:

- a **ports** keyword, and
- within braces, a list of ports.

For example:

```
ports {
```

```
        ...
}
```

Sender Port

The sender port is a provider port, its declaration consists of:

- a **sender** keyword
- the sender port short name
- the **provides** keyword
- the reference to the associated sender receiver interface or mode switch interface
- an optional list of communication specification declarations enclosed within braces, each communication specification (comSpec) consists of:
    - the **queuedComspec** or the **unqueuedComSpec** keyword
    - a reference to a data element belonging to the sender receiver interface declared for this port.
    - (optional) for an **unqueuedComspec** declaration:
        - specifying the **canInvalidate** keyword indicates that the component can actively invalidate data.
        - specifying the **initValue** keyword specifies the value to be used in case the sending component is not yet initialized. If the sender also specifies an initial value the receiver's value will be used. The value is a reference to a value specification, i.e. can be integer, boolean, constant reference etc.
        - specifying the **usesEndToEndProtection** flag indicates that the corresponding data element shall be transmitted using end-to-end protection.
        - (Mandatory for AUTOSAR 4x) specifying the **handleOutOfRange** controls how values that are out of the specified range are handled according to the values of HandleOutOfRangeEnum. Valid values are: **NONE**, **IGNORE**, **SATURATE**, **DEFAULT**, **INVALID**. (Note this field is mandatory in AUTOSAR 4x, but is not supported in previous AUTOSAR releases)
    - (optional) for a **queuedComSpec** declaration:
        - specifying the **usesEndToEndProtection** flag indicates that the corresponding data element shall be transmitted using end-to-end protection.
        - specifying the **handleOutOfRange** controls how values that are out of the specified range are handled according to the values of HandleOutOfRangeEnum. Valid values are: **NONE**, **IGNORE**, **SATURATE**, **DEFAULT**, **INVALID**.

For example:

```
sender mySenderPort provides mySRInterface {
        queuedComSpec element1
        unqueuedComSpec element1 canInvalidate initValue refConstant
}
```

Receiver Port

The receiver port declaration consists of:

- a **receiver** keyword
- the receiver port short name
- the **requires** keyword
- the reference to the associated sender receiver interface or mode switch interface
- an optional list of communication specification (comSpec) declarations enclosed within braces, there are two varieties of comSpecs that can be declared: queued and unqueued communication specifications.
- Queued communication specifications consist of:
    - the **queuedComSpec** keyword
    - the communication specification short name
    - (optional) opening brace
    - the **queueLength** keyword
    - the length of the queue for received events, an integer
    - (optional) the **usesEndToEndProtection** flag indicates that the corresponding data element shall be transmitted using end-to-end protection.
    - (optional) the **handleOutOfRange** controls how values that are out of the specified range are handled

- (optional) the **handleOutOfRange** controls how values that are out of the specified range are handled according to the values of HandleOutOfRangeEnum. Valid values are: **NONE**, **IGNORE**, **SATURATE**, **DEFAULT**, **INVALID**.
    - (optional) closing brace.

- Unqueued communication specifications consist of:
    - the **unqueuedComSpec** keyword
    - the communication specification short name
    - (optional) opening brace
    - the **aliveTimeOut** keyword followed by its double value. The *alive timeout* specifies the amount of time (in seconds) after which the software component (via the RTE) needs to be notified when the corresponding data item has not been received. This is according to the specified timing description.
    - (optional) the **resyncTime** keyword followed by its double value. The *resync time* is the time allowed for resynchronisation of data values when the current data is lost, for example after an ECU reset.
    - (optional) the **handleInvalidType** keyword followed by the value (either 'keep' or 'replace'). This specifies the strategy for handling the reception of an invalidValue.
    - (optional) the **initValue** keyword followed by a value specification
    - (optional) the **enableUpdate** keyword followed by a boolean value
    - (optional) the **usesEndToEndProtection** flag indicates that the corresponding data element shall be transmitted using end-to-end protection.
    - (optional) the **handleOutOfRange** controls how values that are out of the specified range are handled according to the values of HandleOutOfRangeEnum. Valid values are: **NONE**, **IGNORE**, **SATURATE**, **DEFAULT**, **INVALID**.
    - (optional) the **handleNeverReceived** flag indicates whether for the corresponding VariableDataPrototype the "never received" flag is available.
    - (optional) closing brace.

For example:

```
receiver myReceiverPort requires mySRInterface {
        queuedComSpec element1 {
                queueLength 20
        }

        queuedComSpec element2 queueLength 20

        unqueuedComSpec element3 aliveTimeOut 5.0 resyncTime 3.9 handleInvalidType keep initValue refConstant

        unqueuedComSpec element4 {
                aliveTimeOut 5.0
                resyncTime 3.9
                handleInvalidType keep
                initValue refConstant
        }
}
```

SenderReceiverPort

The senderReceiver port declaration consists of:

- a **senderReceiver** keyword
- the senderReceiver port short name
- the **requiresAndProvides** keyword
- the reference to the associated sender receiver interface or mode switch interface
- an optional list of communication specification (comSpec) declarations. The senderReceiver port can have both sending port and receiver port communication specifications
    - All sender communication specifications are enclosed within **senderSpecs** keyword and its braces
    - All receiver communication specifications are enclosed within **receiverSpecs** keyword and its braces

For example:

```
senderReceiver mySenderReceiverPort requiresAndProvides mySRInterface {
        receiverSpecs {
                queuedComSpec element1 {
```

```
                queueLength 20
        }
        queuedComSpec element2 queueLength 20

        unqueuedComSpec element3 aliveTimeOut 5.0 resyncTime 3.9 handleInvalidType keep initValue refConstant

        unqueuedComSpec element4 {
                aliveTimeOut 5.0
                resyncTime 3.9
                handleInvalidType keep
                initValue refConstant
        }
    }
    senderSpecs {
        queuedComSpec element5
        unqueuedComSpec element6 canInvalidate initValue refConstant
    }
}
```

Network Representation

The networkRepresentation object can be added as an optional keyword for

- unqueuedComSpec
- queuedComSpec

The networkRepresentation object can be used for both sender and receiver comSpecs.

The networkRepresentation object declaration consists of

- a **networkRepresentation** keyword
- an optional list of NetworkRepresentation elements.
    - (Optional) the **implDataType** keyword followed by a reference to the implementation data type name
    - (Optional) the **baseType** keyword followed by a reference to the base type name
    - (Optional) the **compuMethod** keyword followed by a reference to the CompuMethod name
    - (Optional) the **dataConstr** keyword followed by a reference to the DataConstr name

For example:

```
    queuedComSpec element2 queueLength 0 handleOutOfRange DEFAULT networkRepresentation {
                    implDataType myIntegerImpl
                    baseType myBaseType
                    compuMethod myCompu
                    dataConstr myIntegerDataConstr
                }

    unqueuedComSpec element3 handleOutOfRange NONE networkRepresentation {
                implDataType myIntegerImpl
                    baseType myBaseType
                    compuMethod myCompu
                    dataConstr myIntegerDataConstr
                }
```

Client Port

The client port declaration consists of:

- the **client** keyword
- the client port short name
- the **requires** keyword
- the reference to the associated client server interface
- an optional list of communication specification (comSpec) declarations enclosed within braces, each comSpec consists of:
    - the **comSpec** keyword
    - a reference to an operation defined in the associated client service interface.

For example:

```
client myClientPort requires myCSInterface {
        comSpec operation1
        comSpec operation2
}
```

Server Port

The server port declaration consists of:

- the **server** keyword
- the server port short name
- the **provides** keyword
- the reference to the associated client server interface
- an optional list of communication specification (comSpec) declarations enclosed within braces, each comSpec consists of:
    - the **comSpec** keyword
    - a reference to an operation defined in the associated client service interface. This specifies the operation that the communication attributes apply to.
    - (optional) opening brace
    - the **queueLength** keyword
    - the length of the call queue on the server side, an integer. The queue is implemented by the RTE
    - (optional) closing brace.

For example:

```
server myServerPort provides myCSInterface {
        comSpec operation1 queueLength 20
        comSpec operation2 {
                queueLength 10
        }
}
```

ServerClient Port

The serverClient port declaration consists of:

- the **serverClient** keyword
- the serverClient port short name
- the **requiresAndProvides** keyword
- the reference to the associated client server interface
- an optional list of communication specification (comSpec) declarations. The serverClient port can have both server port and client port communication specifications
    - All server communication specifications are enclosed within **serverSpecs** keyword and its braces
    - All client communication specifications are enclosed within **clientSpecs** keyword and its braces

For example:

```
serverClient myServerClientPort requiresAndProvides myCSInterface {
        serverSpecs {
                comSpec operation1 queueLength 20
                comSpec operation2 {
                        queueLength 10
                }
        }
        clientSpecs {
                comSpec operation3
                comSpec operation4
        }
}
```

Parameter PPort

A parameter represents an access point to a parameter interface. The declaration consists of:

- the **param** keyword
- the parameter port short name
- the **provides** keyword
- the reference to the associated param interface

For example:

```
param myParameterPPort provides aParamInterface
```

Parameter RPort

A parameter represents the access to a parameter interface (former calPrm interface). The declaration consists of:

- the **param** keyword
- the parameter port short name
- the **requires** keyword
- the reference to the associated param interface

For example:

```
param myParameterRPort requires aParamInterface
```

Parameter PRPort

A parameter represents the access to a parameter interface (former calPrm interface). The declaration consists of:

- the **param** keyword
- the parameter port short name
- the **requiresAndProvides** keyword
- the reference to the associated param interface

For example:

```
param myParameterPRPort requiresAndProvides aParamInterface
```

Internal Behavior

The internal behavior of an application software component describes the runtime environment aspects of a component, that is the runnable entities and the events they respond to.

The declaration consists of:

- the **internalBehavior** keyword
- optional **supportsMultipleInstantiation** flag that indicates that a component can be multiply instantiated which will result in an appropriate component API on programming language level (with or without instance handle)
- the short name of the internal behavior
- the **for** keyword followed by a reference to a SoftwareComponentType
- within braces (each item will be explained in more detail in the following sections) :
    - a list of data type mapping sets
    - a list of included data type sets
    - a list of exclusive ares
    - a list of inter-runnable variables
    - a list of per instance calibration parameter elements
    - a list of shared calibration parameter elements
    - a list of per instance memories (this is only required if this behavior supports multiple instantiation)
    - a list of port API options

- a list of port API options
- a list of runnable entities

For example:

```
internalBehavior supportsMultipleInstantiation name for mySoftwareComponent {
        ...
}
```

Included Data Type Sets

An included data type set declares that a set of AUTOSAR data types are used for the C / C++ implementation of the software component. The AUTOSAR data types become part of the contract. This feature is only available for AUTOSAR 4.0.

The declaration consists of:

- the **includedDataTypeSet** keyword
- optional **literalPrefix** keyword followed by the literal prefix string
- within curly braces:
    - a list of AUTOSAR data type references these are separated with a space.

For example:

```
...
internalBehavior testInternalBehavior for myComponent {
        includedDataTypeSet literalPrefix "literalPrefix" {
                myBoolean
                myString
        }
        includedDataTypeSet {
                myReal
                myFixedPoint
                myBooleanArray
        }
        ...
}
...
```

Data Type Mapping Sets

Data type mappings sets allow code generators to match a specific application data type to an implementation type. This configuration option is only available for AUTOSAR 4.0 models.

The declaration consists of:

- the **dataTypeMappings** keyword
- a list of references to data type mapping sets.

For example:

```
internalBehavior name {
        dataTypeMappings {
                myDataTypeMappingSet1
                myDataTypeMappingSet2
        }
        ...
}
```

Exclusive Areas

Exclusive Areas only allow one Runnable Entity to execute at a time from the list of associated Runnable Entities. This enforces a constraint on certain Runnable Entities that they may never be run concurrently.

The declaration consists of:

- the **exclusiveArea** keyword
- the short name of the exclusive area

For example:

```
internalBehavior supportsMultipleInstantiation name {
        exclusiveArea myExclusiveAreaName
        ...
}
```

Inter-runnable Variables

Implement state message semantics for establishing communication among runnables of the same component.

The declaration consists of:

- the **var** keyword
- the type of the variable
- the optional communication approach, being either **explicit** or **implicit** (default is implicit)
- the short name of the inter-runnable variable
- an optional declaration of the inital value (for value specification syntax see ConstantSpecification).

> The initial value of an inter-runnable variable can either be specified by referencing an existing constant or by specifying the initial value directly. For AUTOSAR releases 2.1 and 3.x, when specifying the initial value directly, a constant specification with the given value will be generated automatically.

For example:

```
const myBoolean TRUE_CONST = true

internalBehavior supportsMultipleInstantiation name {
        ...
        var myBoolean explicit myVariable
        var myBoolean implicit myVariable
        var myBoolean myVariable // implicit
        var myBoolean implicit myVariable = TRUE_CONST // with reference to constant

        // generates constant myVariableInitialValue for AUTOSAR 2.1 & 3.x
        var myBoolean implicit myVariable = true
        ...
}
```

Parameters per instance or shared

The declaration consists of:

- the **instanceParam** or **sharedParam** keyword
- a reference to the associated data type
- the short name of the calibration parameter
- optional category supplied as a string
- optional keyword **initValue** followed by an initial value for the parameter.

For example:

```
internalBehavior supportsMultipleInstantiation name {
        ...
        instanceParam myBoolean myParamElement1 "category1"
        sharedParam myString myParamElement2 initValue "myInitValue"
        ...
}
```

Per instance memories

Software components that support multiple instantiation (attribute "supportsMultipleInstantiation" is true) will typically need memory per instance. It is the responsibility of the RTE to provide a mechanisms with which each instance of a software-component can access its own instance-specific memory.
A Software component can define an arbitrary number of per-instance memory blocks.
For each such memory block, the software component must provide the name of the type (the "C"-type), that it wants to store in the memory block. This attribute allows the RTE to generate an API that provides a convenient and type-safe access.
In addition, the software-component must define the "type" in the attribute "typeDefinition". This attribute must contain a "C" typedef of the "type" in valid C-syntax. This "typeDefinition" must be such that it can be included verbatim in a C header file.
Note that the per-instance memory is not explicitly initialized by the RTE. It is the responsibility of the SW-Component to initialize the per-instance memory.
More details on the use of these attributes in the generation of component header-files can be found in the RTE specification.
Software components that do NOT support multiple instantiation (attribute "sup-portsMultipleInstantiation" is FALSE) do not need to use the "PerInstanceMemory": because there will only be a single instance of the software-component on an ECU, the software-component can use static variables to store the component's state.

perInstanceMemory shortName type typeDefinition

The declaration consists of:

- the **perInstanceMemory** keyword
- the short name of the per instance memory
- the type which is a valid C type that is to be stored in the memory block
- the type definition which is a valid C typedef of the type in valid C syntax.

For example:

```
internalBehavior supportsMultipleInstantiation name {
        ...
        perInstanceMemory myInstanceMemory "countrycode_uint16" "uint16"
        ...
}
```

Port API Options

These port API options dictate how the call signatures for application software component types are generated. This relates to communication over ports, in both directions - i.e. calls to runnables, and from a runnable to a port.

The declaration consists of:

- the **portAPIOption** keyword
- an optional **indirectAPI** keyword - this option switches the generation of the RTE's indirect API functionality for a certain PortPrototype
- an optional **enableTakeAddress** keyword (from AUTOSAR 3.x onwards) - if supplied indicates that the generated API related to this PortPrototype is provided in a way that the software-component is able to use the API reference for deriving an pointer to an object
- a reference to the port
- optional : within braces a list of the port argument values consisting of:
    - reference to a data type (implementation data type in 4.0)
    - corresponding value consistent with the referenced data type
    - a comma separates the list of port argument values.

For example:

```
portAPIOption indirectAPI enableTakeAddress refToAPort {
        myInteger 100,
        myString "abc",
        myBool true
}
```

Runnable Entities

Runnable entities (represented by the meta-class "RunnableEntity") are the smallest code-fragments that are provided by the component and are (at least indirectly) a subject for scheduling for the underlying operating system. Runnables are created to respond to data receipt or operation invocation on a server.

The declaration consists of:

- the **runnable** keyword
- an optional **concurrent** keyword, if present, indicates that the runnable can be invoked concurrently. This implies that the implementation is responsible for the management of this concurrent runnable. Otherwise, the absence of this keyword indicates that the runnable cannot be invoked concurrently.
- the short name of the runnable
- within square brackets the minimum start interval which is of type double
- optional declaration of *inside exclusive areas* representing the referenced exclusive areas that this runnable entity runs inside of. The declaration consists of:
    - the **in** keyword
    - a comma separated list of exclusive area references
- an optional declaration of *uses exclusive areas*, this defines the referenced exclusive areas that this runnable can enter/leave through explicit API calls. Its declaration consist of:
    - the **uses** keyword
    - a comma separated list of exclusive area references
- enclosed within braces (note each will be described in more detail in the following sections):
    - the optional **symbol** keyword
        - the symbol describing this runnable's entry point. This is considered the API of the runnable and is required during the RTE contract phase. By default a symbol with the same name as the runnable will be generated.
    - a list of read variables
    - a list of written variables
    - a list of run time environment events which trigger this runnable
    - a list of parameter accesses
    - a list of data read accesses, which specify the data elements that are only to be read
    - a list of data receive points, which specify the data elements whose value is copied into a buffer for exclusive access, then back to its original location
    - a list of data send points, which specify the data elements that may have their value changed several times, but have their final value written to the receiver point
    - a list of data write accesses, specifies the data elements that are only written
    - a list of mode switch points
    - a list of server call points
    - a list of wait points, which specify the maximum time a runnable blocks/waits for one of the events in its list to occur

For example:

```
runnable concurrent myRunnableName[1.0] in refExclusiveArea1, refExclusiveArea2 uses refExclusiveArea3 {
        symbol "mySymbol"
        ...
}
```

Read and Written Variables

Read variables represent inter-runnable variables that are read by this Runnable.
Written variables signify inter-runnable variables that are written by Runnable.

The declaration consists of:

- the **readVariables** or the **writtenVariables** keyword
- within braces a comma separated list of inter-runnable variable references.

For example:

```
runnable concurrent myRunnableName[1.0] in refExclusiveArea1, refExclusiveArea2 uses refExclusiveArea3 {
        symbol "mySymbol"

        readVariables {
                refInterRunnableVar1,
                refInterRunnableVar2,
                refInterRunnableVar3
        }


        writtenVariables {
                refInterRunnableVar4
        }

        ...
}
```

Runtime Environment Events

Asynchronous Server Call Returns Event

This event is supported in all AUTOSAR releases.
For AUTOSAR 4x, an AsynchronousServerCallResultPoint element will be automatically created and added to the associated Runnable Entity. This element references an AsychronousServerCallPoint.

The declaration consists of:

- the **asynchronousServerCallReturnsEvent** keyword
- a reference to an asynchronous server call point
- optional **as** keyword, followed by the short name for the event
    - If no name is supplied, then a short name will be generated: **asynchronousServerCallReturnsEvent_[** ***referenced server call point short name*** **]** .

For example:

```
internalBehavior supportsMultipleInstantiation myInternalBehavior for component1{
        runnable concurrent runnableName [1.0]  {
                serverCallPoint asynchronous timeOut 1.0 rPortClient.operation11 as myAsyncServerCallPoint
                asynchronousServerCallReturnsEvent myAsyncServerCallPoint as eventShortName
        }
}
```

Data Write Completed Event

This event is only supported from AUTOSAR 4x onwards.

The declaration consists of:

- a **dataWriteCompletedEvent** keyword
- followed by a reference to a data write access variable
- optional **as** keyword, followed by the short name for the event
    - if no short name is given, then one will be generated using the following format: **dataWriteCompletedEvent_[** ***reference to data write access short name*** **]** .

For example:

```
internalBehavior supportsMultipleInstantiation myInternalBehavior for component1{
        runnable concurrent runnableName [1.0]  {
                dataWriteAccess pPortProvider.myDataElement as myDataWriteAccess
                dataWriteCompletedEvent myDataWriteAccess as eventShortName
        }
}
```

Data Send Completed Event

This event is raised when the referenced data elements have been sent or an error occurs. This event will start the runnable in which it belongs to.

The declaration consists of:

- the **dataSendCompletedEvent** keyword
- a reference an existing data send point.
- the optional short name of the event after the **as** keyword

For example:

```
dataSendCompletedEvent refDataSendPoint as shortName
```

Data Received Event

This event is raised when the referenced data elements are received. This event will start the runnable in which it belongs to.

The declaration consists of:

- the **dataReceivedEvent** keyword
- a reference to the receiverPort
- a '.' (dot) character
- followed by a reference to the associated data element.
- the optional short name of the event after the **as** keyword

For example:

```
dataReceivedEvent receiverPortRef.dataElementRef as shortName
```

Receive Error Event

This event is fired when an error occurs in receiving data. This event will start the runnable in which it belongs to.

The declaration consists of:

- the **dataReceiveErrorEvent** keyword
- a reference to the receiverPort
- a '.' (dot) character
- followed by a reference to the associated data element.
- the optional short name of the event after the **as** keyword

For example:

```
dataReceiveErrorEvent  receiverPortRef.dataElementRef as shortName
```

Timing Event

A TimingEvent is required for runnables that need to be executed periodically. This event will start the runnable in which it belongs to.

The declaration consists of:

- the **timingEvent** keyword

- followed by the period in seconds.
- the optional short name of the event after the **as** keyword

For example:

```
timingEvent 1.0 as shortName
```

Mode Switch Event

This event is fired to start a particular runnable when a specific mode is entered or exited. This event will start the runnable in which it belongs to.

The declaration consists of:

- the **modeSwitchEvent** keyword
- the activation kind, being either **entry** or **exit**
- a reference comprising of the requester port
- a '.' character
- a reference to a mode declaration group
- a '.' character
- and a reference to the mode declaration.
- the optional short name of the event after the **as** keyword

For example:

```
modeSwitchEvent entry refRPort.refModeGroup.refMode1 as shortName
```

Operation Invokved Event

A software-component can define an *OperationInvokedEvent* for each operation inside one of the server Provider Ports. This way a Runnable may respond to such an invocation through the generic event handling mechanisms.

The declaration consists of:

- the **operationInvokedEvent** keyword
- a reference to a provider port
- a '.' character
- followed by a reference to an operation associated with that provider port.
- the optional short name of the event after the **as** keyword

For example:

```
operationInvokedEvent pPort.myOperation as myOpEvent
```

Mode disabling dependency

All RTE Events can specify one or more mode disabling dependencies. For example:

```
...
timingEvent 1.0 as myTimingEvent disabledFor {
                myRPort.myModeGroupPrototype.myMode
}
dataReceivedEvent receiverPortRef.dataElementRef  disabledFor {
                myRPort.myModeGroupPrototype.myMode, myRPort.myModeGroupPrototype.myOtherMode
}
...
```

Parameter Access (former Calibration Parameter Access)

This element appears within the braces of a Runnable Entity declaration. There can also be multiple declarations of these elements.

The declaration consists of:

- the **paramAccess** keyword
- a reference to a calibration parameter element prototype, which is defined as either:
    - a shared parameter (which must be defined in the enclosing internal behavior)
    - a per instance parameter (which must be defined in the enclosing internal behavior)
    - an imported parameter via a parameter port. The port must be declared explicitly.
- optional short name declared by the **as** keyword followed by the short name.

If a parameter port is supplied, then the star "*" notation may be used to create paramAccess for all parameter prototypes of the referenced port.

- optional (only with the star notation) **usePrefix** keyword to specify a prefix for the generated assembly connectors
- followed by the prefix as a string.

Note: if the short name is not explicitly provided, then the default name will be composed of:
paramAccess_[***the runnable name*****] _[** ***the referenced calibration parameter element's short name*****]**

For example:

```
paramAccess refParamElementName as myParamAccess1
paramAccess refParamElementName2
paramAccess paramPort.refParamElementName3
paramAccess paramPort.*
paramAccess paramPort2.* usePrefix "myParamAccessPrefix"
```

Data Read Access

The presences of a DataReadAccess means that the runnable needs access to the dataElement in the requester port.
The runnable will not modify the contents of the data but only read the information.
The runnable expects that the contents of this data does NOT change during its entire execution.
This element appears within the braces of a Runnable Entity declaration. There can also be multiple declarations of these elements.

The declaration consists of:

- the **dataReadAccess** keyword
- a reference to a requester port short name
- a '.' then
- to explicitly declare the data element
    - a reference to the data element that belongs to the interface associated with the port
    - optional declaration of the short name, defined by specifying the **as** keyword followed by the short name.
- or to connect all data element prototypes belonging to the referenced interface, use the star "*" notation
    - optional (only with the star notation) **usePrefix** keyword to specify a prefix for the generated assembly connectors
    - followed by the prefix as a string.

Note: if the short name is not explicitly provided, then the default name will be composed of:
dataReadAccess_[***the runnable's short name*****] _[** ***the requester port's short name*****] _[** ***the referenced data element's short name*****]**

For example:

```
dataReadAccess refRPortName.refDataElement as myAccessName
dataReadAccess refRPortName2.refDataElement2
dataReadAccess refRPortName.*
dataReadAccess refRPortName2.* usePrefix "myDataReadAccessPrefix"
```

Data Receive Point

A data receive point allows a runnable to explicitly query for received information, thereby blocking write access to the same information only for a very brief period.

This element appears within the braces of a Runnable Entity declaration. There can also be multiple declarations of these elements.

The declaration consists of:

- the **dataReceivePoint** keyword
- a reference to a receiver port
- a '.' then
- to explicitly declare the data element
    - a reference to the data element that belongs to the interface associated with the port
    - optional declaration of the short name, defined by specifying the **as** keyword followed by the short name.
- or to connect all data element prototypes belonging to the referenced interface, use the star "*" notation
    - optional (only with the star notation) **usePrefix** keyword to specify a prefix for the generated assembly connectors
    - followed by the prefix as a string.

Note: if the short name is not explicitly provided, then the default name will be composed of:
dataReceivePoint_[***the runnable's short name*****] _[** ***the receiver port's short name*****] _[** ***the referenced data element's short name*****]**

For example:

```
dataReceivePoint refRPort.refDataElement as myPointName
dataReceivePoint refRPort2.refDataElement2
dataReceivePoint refRPort.*
dataReceivePoint refRPort2.* usePrefix "myDataReceivePointPrefix"
```

Data Send Point

A data send point specifies that a runnable explicitly sends a certain data element.

This element appears within the braces of a Runnable Entity declaration. There can also be multiple declarations of these elements.

The declaration consists of:

- the **dataSendPoint**
- a reference to a provider port
- a '.'
- to explicitly declare the data element
    - a reference to the data element that belongs to the interface associated with the port
    - optional declaration of the short name, defined by specifying the **as** keyword followed by the short name.
- or to connect all data element prototypes belonging to the referenced interface, use the start "*" notation
    - optional (only with the star notation) **usePrefix** keyword to specify a prefix for the generated assembly connectors
    - followed by the prefix as a string.

Note: if the short name is not explicitly provided, then the default name will be composed of:
dataSendPoint_[***the runnable's short name*****] _[** ***the provider port's short name*****] _[** ***the referenced data element's short name*****]**

For example:

```
dataSendPoint  refPPort.refDataElement as myPointName
dataSendPoint  refPPort2.refDataElement2
dataSendPoint  refPPort2.*
```

```
dataSendPoint refPPort.* usePrefix "myDataSendPointPrefix"
```

Data Write Access

The presences of a DataWriteAccess means that the runnable will potentially modify the dataElement in the pPort. The runnable has free access to the dataElement while it is running. The runnable has the responsibility to make sure that the dataElement is in a consistent state when it returns. When using DataWriteAccess the new values of the dataElement is not made available via the communication infrastructure before the runnable returns (exits the "Running" state).

This element appears within the braces of a Runnable Entity declaration. There can also be multiple declarations of these elements.

The declaration consists of:

- the **dataWriteAccess** keyword
- a reference to the provider port
- a '.' then

either:

-
  - a reference to the data element that belongs to the interface associated with the port
  - optional declaration of the short name, defined by specifying the **as** keyword followed by the short name.

or:

- to connect all data element prototypes belonging to the referenced interface, use the star "*" notation
  - optional (only with the star notation) **usePrefix** keyword to specify a prefix for the generated assembly connectors
  - followed by the prefix as a string.

Note: if the short name is not explicitly provided, then the default name will be composed of:
dataWriteAccess_[*the runnable's short name*] _[*the provider port's short name*] _[*the referenced data element's short name*]

For example:

```
dataWriteAccess refPPort.refDataElement as myPointName
dataWriteAccess refPPort2.refDataElement2
dataWriteAccess refPPort.*
dataWriteAccess refPPort2.* usePrefix "myDataWriteAccessPrefix"
```

Mode Switch Point

This element appears within the braces of a Runnable Entity declaration. There can also be multiple declarations of these elements.

The declaration consists of:

- the **modeSwitchPoint** keyword
- a reference to a provider port
- a '.' then
- a reference to the mode group prototype
- optional declaration of the short name, defined by specifying the **as** keyword followed by the short name.

Note: if the short name is not explicitly provided, then the default name will be composed of:
modeSwitchPoint_[*the runnable's short name*] _[*the provider port's short name*] _[*referenced mode group prototype's short name*]

For example:

```
modeSwitchPoint refProviderPort.refModeGroupPrototype as myModeSwitchPoint
modeSwitchPoint refProviderPort.refModeGroupPrototype2
```

ModeAccessPoint

This event is fired to start a particular runnable when a specific mode is entered or exited. This event will start the runnable in which it belongs to.

The declaration consists of:

- the **modeAccessPoint** keyword
- a reference comprising of the requester or provider port
- a '.' character
- a reference to a mode declaration group
- then optionally, a **as** keyword followed by the short name.

For example:

```
modeAccessPoint refRPort.refModeGroup as shortName
```

> Specifying mode access points is only required when transforming the model to AUTOSAR 4.0. However, we still recommend declaring mode access points for all releases for documentation purposes as well as to ease future migration to AUTOSAR 4.0.

Server Call Point

When a runnable has a server call point, it has the possibility to invoke any of the operations of a specific requester port of the component.

The declaration consists of:

- the **serverCallPoint** keyword
- the **asynchronous** or **synchronous** keyword
- the **timeOut** keyword which is the time in seconds before the server call times out and returns with an error message. It depends on the call type (synchronous or asynchronous) how this is reported. If a timeOut is not supplied, the default value will be set to 0.0.

either:

- a reference to a requester port, followed by a '.' then a reference to the operation associated to the port
- then optionally, a **as** keyword followed by the short name

or:

- to connect all operations belonging to the referenced interface, use the star "*" notation
    - optional (only with the star notation) **usePrefix** keyword to specify a prefix for the generated assembly connectors
    - followed by the prefix as a string.

Note: if the short name is not explicitly provided, then the default name will be composed of:
serverCallPoint_[***the runnable's short name*****] _[** ***the requested port's short name*****] _[** ***the referenced operation's short name*****]**

For example:

```
serverCallPoint asynchronous timeOut 1.0 myRequesterPort.myOperation as myPoint
serverCallPoint asynchronous timeOut 1.0 myRequesterPort.*
serverCallPoint asynchronous timeOut 1.0 myRequesterPort2.* usePrefix "myServerCallPointPrefix"
```

Wait Point

Wait points define the amount of time a runnable can wait until an event in its set of events occurs.
A wait point allows a runnable to block its execution while waiting for an event to be set.

The declaration consists of:

- the **waitPoint** keyword
- the short name

- the **timeOut** keyword
- the time in seconds before the wait point times out and the blocking wait call returns with an error indicating the timeout.
- the keyword **for**
- a comma separated list of run time environment events that this wait point is waiting for.

For example:

```
waitPoint myWaitPoint timeOut 2.0 for refEvent1, refEvent2, refEvent3
```

Implementation

While AUTOSAR contains various component types, only application software components possess an implementation. In the meta model this means that implementations can be given for 'ApplicationSoftwareComponentType' or derived classes only.
The implementation serves the following main purposes, to:

- identify application software component that is implemented
- link to code (source code, object code, ...)
- specify the required build environment (optional)
- specify the compatible runtime environment (software, hardware, resources)

The declaration consists of:

- the **implementation** keyword
- the short name
- the **for** keyword followed by a reference to an InternalBehavior
- the **language** keyword
- the language being either **c**, **cpp** or **java**. This is specifies the programming language the implementation was created with.
- the **codeDescriptor** keyword
- the code descriptor declared as a string.
- (optional) the **codeGenerator** keyword followed by its string value
- (optional) the **requiredRTEVendor** followed by its string value. *NB This information is potentially important at the time of integrating (in particular: linking) the application code with the RTE. The semantics is that (if the association exists) the corresponding code has been created to fit to the vendor-mode RTE provided by this specific vendor. Attempting to integrate the code with another RTE generated in vendor mode is in general not possible.*
    - the **swVersion** keyword followed by its long value
    - the **vendorId** keyword followed by its long value
- (optional) a number of compiler declarations consisting of:
    - **compiler** keyword, followed by its short name
    - the **vendor** keyword followed by its string value
    - the **version** keyword followed by its string value

```
implementation myImplementation for myInternalBehavior {
        language java
        codeDescriptor "src"
        codeGenerator "myGenerator"
        requiredRTEVendor "myRTEVendor"
        compiler myCompiler1 vendor "Vendor1" version "Version1"
        compiler myCompiler2 vendor "Vendor2" version "Version2"
}
```

Note: ResourceConsumption Currently, the SWCD language does not support the explicit definition of resource consumption elements. However, in the background, a placeholder Resource Consumption element is generated, together with a generated shortName for each Implementation. This is to ensure that the model created by SWCD is compliant with AUTOSAR.

Mode Declaration Groups

To declare mode declaration groups along with its mode declarations, specify:

- the **modeGroup** keyword
- the mode declaration group's short name
- the optional keyword **initial** followed by a reference to the initial mode
- a list of comma separated mode declarations enclosed within braces

For example:

```
modeGroup myModeGroup {
        SHUTDOWN, SLEEP, WAKE_SLEEP, STARTUP, RUN, POST_RUN
}

modeGroup myModeGroup2 initial STARTUP {
        SHUTDOWN, SLEEP, WAKE_SLEEP, STARTUP, RUN, POST_RUN
}
```

Compositions

The purpose of an AUTOSAR composition is to allow encapsulation of functionality by aggregating existing software components.

A composition type declaration consists of:

- the **composition** keyword
- the composition short name
- a list of component prototypes and or assembly connectors enclosed within braces.

For example:

```
composition compositionName {
        prototype myComponent1 myPrototype1
        prototype myComponent2 myPrototype2
        connect myPrototype1.mySenderPort to myPrototype2.myReceiverPort
        autoConnect myPrototype1 to myPrototype2
}
```

Component Prototypes

A component prototype declaration consists of:

- the **prototype** keyword
- a reference to a component type
- the short name of the component prototype.

```
prototype refToComponentType componentPrototypeName
```

Delegate Ports

A delegate port declaration consists of:

- the **provides** keyword for PPortPrototypes or the **requires** keyword for RPortPrototypes
- a comma separated list of delegated ports, where each port is specified by a component prototype
- the short name of the delegated port.

```
composition system {
        prototype a
        ports{
                // single port delegation
                requires a.receiverPort delegatedReceiverPort
                provides a.senderPort delegatedSenderPort

                // multiple port delegation
                requires a.receiverPort1, a.receiverPort2 delegatedReceiverPort2
                provides a.senderPort1, a.senderPort2 delegatedSenderPort2
```

```
            provides a.senderPort1, a.senderPort2 delegatedSenderPort2
        }
}
```

Assembly Connectors

An assembly connector declaration consists of:

- the **connect** keyword
- a reference to a component prototype, followed by a '.' then the reference to the provider port
- the **to** keyword
- a reference to a component prototype, followed by a '.' then the reference to the requester port
- optional **as** keyword to supply a short name for this assembly connector
- the short name.

Note that if the short name is not supplied, then a name will be generated based on the port names. It will be composed of the provider component prototype name, provider port short name, the requester component prototype name, and requester port short name. For example: "componentPrototypeA_senderPort_to_componentPrototypeB_receiverPort".

```
connect componentPrototype1.pPort to componentPrototype2.rPort
```

Auto connect

This feature generates all assembly connectors between two component prototypes. A connection is made between two component prototypes when a provider port has a matching interface as a requesting port. Two interfaces match if they are identical or compatible as defined in the AUTOSAR standard's comatibility rules.

If, for a provider port, there exists several possible requester ports to connect to, the connection is marked as ambiguous and not generated. This will appear as a validation error message in the editor, indicating which provider ports were in question.

To resolve any ambiguous connections, a manual **connect** statement can be created, if these **connect** statements address all ambiguous ports from the **autoConnect**, then the validation error message will not appear.

An auto connect statement consists of:

- the **autoConnect** keyword
- the first component prototype
- the **to** keyword
- the second component prototype for which assembly connectors are to be created for
- optional **usePrefix** keyword to specify a prefix for the generated assembly connectors
- followed by the prefix as a string.

All connector names will be generated, consistent with the format described for the Assembly Connectors above.

```
autoConnect componentPrototype1 to componentPrototype2
autoConnect componentPrototype3 to componentPrototype4 usePrefix "mySpecialPrefix"
```

Annotations

Descriptions

Descriptions can be added to any element that is of type "Identifiable", that is, the element has a "shortName" attribute.

To add a description, an annotation can be made directly above the element. The description will be stored in the elements's "desc" attribute in an "MIData2" object within the AUTOSAR model.
To declare a description:

- the **@desc** keyword, followed by
- any letters to form the description.

for a multi line description:

- the **@desc** keyword
- within braces:

- any letters to form the description spanning across multiple lines.

For example:

```
@desc Here is a single line description
boolean myBoolean

@desc {Here is a multi line description.
The description can span across
multiple lines.}
int myInteger min 8 max 10
```

Universal Unique Identifiers (UUID)

Each AUTOSAR element that is of type "Identifiable" has either a supplied or generated universal unique identifier(UUID). The generated UUID is derived from the fully qualified name of the element. Therefore, if the short name is modified, the generated UUID will also be modified.

To supply an alternative UUID to the generated one, an annotation can be placed above the element.
Consisting of:

- a **@uuid** keyword
- followed by the universal unique identifier.

For example:

```
@uuid 951dc165-46e0-40d9-8e79-9b35dd952a51
boolean myBooleanType
```

Otherwise, without an explicitly supplied UUID, the value will be generated and its value will visible via the Eclipse properties view.

Quick Assist

Quick assist is a convenience feature integrated into the ARText editor. It provides convenience functions to auto generate certain aspects of the currently selected element.

To invoke the quick assist, ensure the cursor is on the desired element, and invoke the Control Key + 1.
This will bring up a dialog with the possible quick assist functions available for the selected element.