# 22 Communication operations

TTCN‑3 supports *message-based* and *procedure-based* *unicast*, *multicast* and *broadcast* communication. Furthermore, TTCN‑3 allows to examine the top element of incoming port queues and to control the access to ports by means of *controlling operations*. The communication operations and restrictions on their usage are summarized in table 21.

Table 21: Overview of TTCN‑3 communication operations

|  |
| --- |
| Communication operations |
| Communication operation | Keyword | Can be used at message-based ports | Can be used at procedure-based ports |
| Message-based communication |
| Send message  | **send** | Yes |  |
| Receive message | **receive** | Yes |  |
| Trigger on message | **trigger** | Yes |  |
| Procedure-based communication |
| Invoke procedure call | **call** |  | Yes |
| Accept procedure call from remote entity | **getcall** |  | Yes |
| Reply to procedure call from remote entity | **reply** |  | Yes |
| Raise exception (to an accepted call) | **raise** |  | Yes |
| Handle response from a previous call | **getreply** |  | Yes |
| Catch exception (from called entity) | **catch** |  | Yes |
| Examine top element of incoming port queues |
| Check msg/call/exception/reply received | **check** | Yes | Yes |
| **Controlling operations** |
| Clear port queue | **clear** | Yes | Yes |
| Clear queue and enable sending and receiving at a port | **start** | Yes | Yes |
| Disable sending and disallow receiving operations to match at a port | **stop** | Yes | Yes |
| Disable sending and disallow receiving operations to match new messages/calls | **halt** | Yes | Yes |
| Check the state of a port | **checkstate** | Yes | Yes |

## 22.1 The communication mechanisms

This clause explains the principles of TTCN‑3 communication for message-based communication (see clause 22.1.1), for procedure-based communication (see clause 22.1.2), for unicast, multicast, and broadcast communication (see clause 22.1.3), as well as the general format of sending and receiving operations (see clause 22.1.4).

### 22.1.1 Principles of message-based communication

Message-based communication is communication based on an asynchronous message exchange. Message-based communication is non-blocking on the **send** operation, as illustrated in figure 11, where processing in the SENDER continues immediately after the **send** operation occurs. The RECEIVER is blocked on the **receive** operation until it processes the received message.

In addition to the **receive** operation, TTCN‑3 provides a **trigger** operation that filters messages with certain matching criteria from a stream of received messages on a given incoming port. Messages at the top of the queue that do not fulfil the matching criteria are removed from the port without any further action.



Figure 11: Illustration of the asynchronous send and receive

### 22.1.2 Principles of procedure-based communication

The principle of procedure-based communication is to call procedures in remote entities. TTCN‑3 supports *blocking* and *non-blocking* procedure-based communication. Blocking procedure-based communication is blocking on the calling and the called side, whereas non-blocking procedure-based communication is only blocking on the called side. Signatures of procedures that are used for non-blocking procedure-based communication shall be specified according to the rules in clause 13.

The communication scheme of blocking procedure-based communication is shown in figure 12. The CALLER calls a remote procedure in the CALLEE by using the **call** operation. The CALLEE accepts the call by means of a **getcall** operation and reacts by either using a **reply** operation to answer the call or by raising (**raise** operation) an exception. The CALLER handles the reply or exception by using **getreply** or **catch** operations. In figure 12, the blocking of CALLER and CALLEE is indicated by means of dashed lines.



Figure 12: Illustration of blocking procedure-based communication

The communication scheme of non-blocking procedure-based communication is shown in figure 13. The CALLER calls a remote procedure in the CALLEE by using the **call** operation and continues its execution, i.e. does not wait for a reply or exception. The CALLEE accepts the call by means of a **getcall** operation and executes the requested procedure. If the execution is not successful, the CALLEE may raise an exception to inform the CALLER. The CALLER may handle the exception by using a **catch** operation in an **alt** statement. In figure 13, the blocking of the CALLEE until the end of the call handling and possible raise of an exception is indicated by means of a dashed line.



Figure 13: Illustration of non-blocking procedure-based communication

### 22.1.3 Principles of unicast, multicast and broadcast communication

TTCN‑3 supports unicast, multicast and broadcast communication:

* Unicast communication means one sender to one receiver.
* Multicast communication is from one sender to a list of receivers.
* Broadcast communication is from one sender to all receivers (being connected or mapped to the sender).

The terms unicast, multicast and broadcast communication are related to port communication. This means, it is only possible to address one, several or all test components that are connected to the specified port. Unicast, multicast and broadcast can also be used for mapped ports. In this case, one, several or all entities within the SUT can be reached via the specified mapped port.

### 22.1.4 General format of communication operations

Operations such as **send** and **call** are used for the exchange of information among test components and between an SUT and test components. For explaining the general format of these operations, they can be structured into two groups:

a) a test component sends a message (**send** operation), calls a procedure (**call** operation), or replies to an accepted call (**reply** operation) or raises an exception (**raise** operation). These actions are collectively referred to as *sending operations*;

b) a component receives a message (**receive** operation), awaits a message (**trigger** operation),accepts a procedure call (**getcall** operation), receives a reply for a previously called procedure (**getreply** operation) or catches an exception (**catch** operation). These actions are collectively referred to as *receiving operations*.

#### 22.1.4.1 General format of the sending operations

Sending operations consist of a *send* part and, in the case of a blocking procedure-based **call** operation, a *response* and *exception handling* part.

The send part:

* specifies the port at which the specified operation shall take place;
* defines the message or procedure call to be transmitted;
* gives an (optional) address part that uniquely identifies one or more communication partners to which a message, call, reply or exception shall be send.

The port name, operation name and value shall be present in all sending operations. The address part (denoted by the **to** keyword) is optional and need only be specified in cases of one-to-many connections where:

* unicast communication is used and one receiving entity shall be explicitly identified;
* multicast communication is used and a set of receiving entities has to be explicitly identified;
* broadcast communication is used and all entities connected to the specified port have to be addressed.

EXAMPLE 1:

|  |  |
| --- | --- |
| **Send part** | **(Optional) response and exception** |
| Port and operation | Value part | (Optional) address part | handling part |
| MyP1.**send** | (MyVariable + YourVariable - 2) | **to** MyPartner; |  |

Response and exception handling is only needed in cases of procedure-based communication. The response and exception handling part of the **call** operation is optional and is required for cases where the called procedure returns a value or has **out** or **inout** parameters whose values are needed within the calling component and for cases where the called procedure may raise exceptions which need to be handled by the calling component.

The response and exception handling part of the call operation makes use of **getreply** and **catch** operations to provide the required functionality.

EXAMPLE 2:

|  |  |
| --- | --- |
| Send part | (Optional) response and exception handling part |
| Port and operation | Value part | (Optional) address part |  |
| MyP1.**call** | (MyProc:{MyVar1}) |  | { [] MyP1.**getreply**(MyProc:{MyVar2}) {} [] MyP1.**catch**(MyProc, ExceptionOne) {}} |

#### 22.1.4.2 General format of the receiving operations

A receiving operation consists of a *receive* part and an (optional) *assignment* part.

The receive part:

a) specifies the port at which the operation shall take place;

b) defines a matching part which specifies the acceptable input which will match the statement;

c) gives an (optional) address expression that uniquely identifies the communication partner (in case of one‑to‑many connections).

The port name, operation name and value part of all receiving operations shall be present. The identification of the communication partner (denoted by the **from** keyword) is optional and need only be specified in cases of one‑to‑many connections where the receiving entity needs to be explicitly identified.

The assignment part in a receiving operation is optional. For message-based ports it is used when it is required to store received messages. In the case of procedure-based ports it is used for storing the **in** and **inout** parameters of an accepted call, for storing the return value or for storing exceptions. For the assignment part strong typing is required, e.g. the variable used for storing a message shall have the same type as the incoming message.

In addition, the assignment part may also be used to assign the **sender** address of a message, exception, **reply** or **call** to a variable. This is useful for one-to-many connections where, for example, the same message or call can be received from different components, but the message, reply or exception shall be sent back to the original sending component.

For receiving operations using the any port from a port array construction (see clause 22.2.2), the assignment part may also be used to store the indices that identify the specific port instance where the receiving operation matched.

EXAMPLE:

|  |  |  |
| --- | --- | --- |
| Receive part |  | (Optional) assignment part |
| Port and operation | Matching part | (Optional) address expression |  | (Optional) value assignment | (Optional) parameter value assignment | (Optional) sender value assignment |
| MyP1.getreply | (AProc:{?} value 5) |  | -> |  | param (V1) | sender APeer |

|  |  |  |
| --- | --- | --- |
| Receive part |  | (Optional) assignment part |
| Port and operation | Matching part | (Optional) address expression |  | (Optional) value assignment | (Optional) parameter value assignment | (Optional) sender value assignment |
| MyP2.receive | (MyTemplate(5,7)) | from APeer | -> | value MyVar |  |  |

|  |  |  |
| --- | --- | --- |
| Receive part |  | (Optional) assignment part |
| Port and operation | Matching part | (Optional) address expression |  | (Optional) value assignment | (Optional) parameter value assignment | (Optional) sender value assignment | (Optional)port index assignment |
| any from P.receive | (MyTemplate(5,7)) |  | -> |  |  |  | @index **value** I |

## 22.2 Message-based communication

The operations for message-based communication via asynchronous ports are summarized in table 22.

Table 22: Overview of TTCN‑3 message-based communication

|  |  |
| --- | --- |
| Communication operation | Keyword |
| Send message  | **send** |
| Receive message | **receive** |
| Trigger on message | **trigger** |
| Check message received | **check** |

### 22.2.1 The Send operation

The **send** operation is used to place a message on an outgoing message port.

***Syntactical Structure***

*Port* "." **send** "(" *TemplateInstance* ")"

[ **to** *Address* ]

NOTE: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**all component**".

***Semantic Description***

The **send** operation places a message on an outgoing message port. The message may be specified by referencing a defined template or can be defined as an in-line template.

**Sending unicast, multicast or broadcast**

Unicast, multicast and broadcast communication can be determined by the optional **to** clause in the **send** operation. A **to** clause can be omitted in case of a one-to-one connection where unicast communication is used and the message receiver is uniquely determined by the test system structure.

Unicast communication is specified, if the **to** clause addresses one communication partner only. Multicast communication is used, if the **to** clause includes a list of communication partners. Broadcast is defined by using the **to** clause with **all component** keyword.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

a) The TemplateInstance (and all parts of it) shall have a specific value i.e. the use of matching mechanisms such as *AnyValue* is not allowed.

b) When defining the message in-line, the optional type part shall be used if there is ambiguity of the type of the message being sent.

c) The **send** operation shall only be used on message-based ports and the type of the template to be sent shall be in the list of outgoing types of the port type definition.

d) A **to** clause shall be present in case of one-to-many connections.

e) *AddressRef* shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **send** operation. No *AddressRef* shall contain the special value **null** at the time of the operation.

f) Applying a **send** operation to an unmapped or disconnected port shall cause a test case error.

***Examples***

EXAMPLE 1: Simple send (receiver is determined from the test configuration)

 MyPort.**send**(MyTemplate(5,MyVar)); // Sends the template MyTemplate with the actual

 // parameters 5 and MyVar via MyPort.

 MyPort.**send**(5); // Sends the integer value 5 (which is an in-line template)

EXAMPLE 2: Sending with explicit to clause

 MyPort.**send**(**charstring**:"My string") **to** MyPartner;

 // Sends the string "My string" to a component with a

 // component reference stored in variable MyPartner

 MyPCO.**send**(MyVariable + YourVariable - 2) **to** MyPartner;

 // Sends the result of the arithmetic expression to MyPartner.

 MyPCO2.**send**(MyTemplate) **to** (MyPeerOne, MyPeerTwo);

 // Specifies a multicast communication, where the value of

 // MyTemplate is sent to the two component references stored

 // in the variables MyPeerOne and MyPeerTwo.

 MyPCO3.**send**(MyTemplate) **to all component**;

 // Broadcast communication: the value of Mytemplate is send to

 // all components which can be addressed via this port. If

 // MyPCO3 is a mapped port, the components may reside inside

 // the SUT.

### 22.2.2 The Receive operation

The **receive** operation is used to receive a message from an incoming message port queue.

***Syntactical Structure***

( *Port* | **any** **port** | **any from** PortArrayRef ) "." **receive**

[ "(" *TemplateInstance* ")" ]

[ **from** *Address* ]

[ "->" [ **value** ( *VariableRef* |

 ( "(" { *VariableRef* [ ":=" [ **@decoded** [ "(" *Expression* ")" ] ]

 *FieldOrTypeReference* ][","] } ")" )

 ) ]

 [ **sender** *VariableRef* ]

 [ @**index** **value** *VariableRef* ] ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

***Semantic Description***

The **receive** operation is used to receive a message from an incoming message port queue. The message may be specified by referencing a defined template or can be defined as an in-line template.

The **receive** operation removes the top message from the associated incoming port queue if, and only if, that top message satisfies all the matching criteria associated with the **receive** operation.

If the match is not successful, the top message shall not be removed from the port queue i.e. if the **receive** operation is used as an alternative of an **alt** statement and it is not successful, the execution of the test case shall continue with the next alternative of the **alt** statement.

**Matching criteria**

The matching criteria are related to the type and value of the message to be received. The type and value of the message to be received are determined by the argument of the **receive** operation, i.e. may either be derived from the defined template or be specified in-line. An optional type field in the matching criteria to the **receive** operation shall be used to avoid any ambiguity of the type of the value being received.

NOTE 2: Encoding attributes also participate in matching in an implicit way, by preventing the decoder to produce an abstract value from the received message encoded in a different way than specified by the attributes.

**Receiving from a specific sender**

In the case of one-to-many connections the **receive** operation may be restricted to a certain communication partner. This restriction shall be denoted using the **from** keyword.

**Storing the received message and parts of the received message**

If the match is successful, the value removed from the port queue and/or parts of this value can be stored in variables or formal parameters. This is denoted by the symbol '->' and the keyword **value**.

When the keyword **value** is followed by a name of a variable or formal parameter, the whole received message shall be stored in the variable or formal parameter. The variable or formal parameter shall be type compatible with the received message.

When the keyword **value** is followed by an assignment list enframed by a pair of parentheses, the whole received message and/or one or more parts of it can be stored. In a single assignment within the list, on the left hand side of the assignment symbol (":=") a field of the template type shall be referenced, on the right hand side the name of the variable or a formal parameter, in which the value shall be stored. The variable or formal parameter shall be type compatible with the type on the left hand side of the assignment symbol. As a special case the field reference can be absent to indicate that the whole message shall be stored in a variable.

When assigning individual fields of a message, encoded payload fields can be decoded prior to assignment using the **@decoded** modifier. In this case, the referenced field on the right hand sided of the assignment shall be one of the **bitstring**, **hexstring**, **octetstring**, **charstring** or **universal** **charstring** types. It shall be decoded into a value of the same type as the variable on the left hand side of the assignment. Failure of this decoding shall cause a test case error. In case the referenced field is of the **universal** **charstring** type, the **@decoded** clause can contain an optional parameter defining the encoding format. The parameter shall be of the **charstring** type and it shall contain one of the strings allowed for the **decvalue\_unichar** function (specified in clause C.5.4). Any other value shall cause an error. In case the referenced field is not a **universal** **charstring**, the optional parameter shall not be present.

NOTE 3: The model of the behaviour of this implicit decoding is defined in clause B.1.2.9.

NOTE 4: The **@decoded** clause is typically used together with the **decmatch** matching mechanism in the matching part of the receive statement. Since the decoding procedures for assignment and matching are virtually the same, TTCN-3 tools can be optimized in such a way that only one call to the decoder is made when the receiving statement contains both **decmatch** matching mechanism and **@decoded** assignment for the same payload field.

**Storing the sender**

It is also possible to retrieve and store the component reference or address of the sender of a message. This is denoted by the keyword **sender**.

When the message is received on a connected port, only the component reference is stored in the following the **sender** keyword, but the test system shall internally store the component name too, if any (to be used in logging).

**Receive any message**

A **receive** operation with no argument list for the type and value matching criteria of the message to be received shall remove the message on the top of the incoming port queue (if any) if all other matching criteria are fulfilled.

**Receive on any port**

To **receive** a message on any port, use the **any port** keywords.

**Receive on any port from a port array**

To **receive** a message on any port from a specific port array, use the **any from** *PortArrayRef*syntax where PortArrayRefshallbe areference to a port array identifier**.** It is also possible to store the index of a port in a single-dimensional port array at which the operation was successful to a variable of type integer or, in case of multi‑dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching messages, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

**Stand-alone receive**

The **receive** operation can be used as a stand-alone statement in a behaviour description. In this latter case the **receive** operation is considered to be shorthand for an **alt** statement with the **receive** operation as the only alternative.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

a) When defining the message in-line, the optional type part shall be present whenever the type of the message being received is ambiguous.

b) The **receive** operation shall only be used on message-based ports and the type of the value to be received shall be included in the list of incoming types of the port type definition.

c) No binding of the incoming values to the terms of the expression or to the template shall occur.

d) A message received by *receive any message* shall not be stored, i.e. the **value** clause shall not be present.

e) Type mismatch at storing the received value or parts of the received value and storing the sender shall cause an error.

NOTE 5: Type mismatch may happen if the types in the receive part are not present or not compatible to the types in the assignment part.

f) *AddressRef* for retrieving the sending entity shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **receive** operation. No *AddressRef* shall contain the special value **null** at the time of the operation.

g) The *PortArrayRef* shall be a reference to a completely initialized port array.

h) The index redirection shall only be used when the operation is used on an any from port array construct.

i) If the index redirection is used for single-dimensional port arrays, the type of the integer variable shall allow storing the highest index of the respective array.

j) If the index redirection is used for multi-dimensional port arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.

k) If a variable referenced in the **value**, **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **receive** operation i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **receive** operation.

l) If the **receive** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause.

m) When assigning implicitly decoded message fields (by using the **@decoded** modifier) in cases where the value or template to be matched uses the *MatchDecodedContent* (**decmatch**) matching for the field to be stored, the type of the template in the *MatchDecodedContent* matching shall be type-compatible to the type of the variable the decoded field is stored into.

***Examples***

EXAMPLE 1: Basic receive

 MyPort.**receive**(MyTemplate(5, MyVar)); // Matches a message that fulfils the conditions

 // defined by template MyTemplate at port MyPort.

 MyPort.**receive**(A<B); // Matches a Boolean value that depends on the outcome of A<B

 MyPort.**receive**(**integer**:MyVar); // Matches an integer value with the value of MyVar

 // at port MyPort

 MyPort.**receive**(MyVar); // Is an alternative to the previous example

EXAMPLE 2: Receiving from a sender, storing the message, parts of the message or the sender

 **type** MyPayloadType **record** {

 **integer** messageId,

 ContentType content

 }

 **type** MyType2 **record** {

 Header header,

 **octetstring** payload

 }

 **template** MyType MyTemplate := {

 messageId := 42,

 content := ?

 }

 ...

 **var** MyPayloadType MyVar;

 **var** **integer** MyMessageIdVar, MyIntegerVar;

 **var** **charstring** MyCharstringVar;

 **var** **address** MyPeer;

 **var** **octetstring** MyVarOne := '00ff'O;

 MyPort.**receive**(**charstring**:"Hello")**from** MyPeer; // Matches charstring "Hello" from MyPeer

 MyPort.**receive**(MyType:?) -> **value** MyVar; // The value of the received message is

 // assigned to MyVar.

 MyPort.**receive**(MyType:?) -> **value** (MyVar, MyMessageIdVar:= MyType.messageId)

 // The value of the received message is stored in the variable

 // MyVar and the value of the messageId field of the received

 // message is stored in the variable MyMessageIdVar.

 MyPort.**receive**(anytype:?) -> **value** (MyIntegerVar := integer)

 // If the received value is an integer, it is stored in the variable

 // MyIntegerVar, a test case error otherwise.

 MyPort.**receive**(charstring:?) -> **value** (MyCharstringVar)

 // The received value is stored in the variable MyCharstringVar;

 // Note that it is the same as to write "**value** MyCharstringVar"

 MyPort.**receive**(A<B) -> **sender** MyPeer; // The address of the sender is assigned to MyPeer

 MyPort.**receive**(MyType:{5, MyVarOne}) -> **value** MyVar **sender** MyPeer;

 // The received message value is stored in MyVarTwo and the sender address is stored in MyPeer.

 MyPort.**receive**(MyType2:{header := ?, payload := **decmatch** MyTemplate }) -> **value** (MyVar := **@decoded** payload);

 // The encoded payload field of the received message is decoded and matched with

 // MyTemplate; if the matching is successful the decoded payload is stored in MyVar.

EXAMPLE 3: Receive any message

 MyPort.**receive**; // Removes the top value from MyPort.

 MyPort.**receive** **from** MyPeer; // Removes the top message from MyPort if its sender is
 MyPeer

 MyPort.**receive** -> **sender** MySenderVar; // Removes the top message from MyPort and assigns

 // the sender address to MySenderVar

EXAMPLE 4: Receive on any port

 **any port**.**receive**(MyMessage);

EXAMPLE 5: Receive on any port from a port array

 **type** **port** MyPort **message** { **inout** **integer** }

 **type** **component** MyComponent {

 **port** MyPort p[10][10];

 }

 **var** **integer** i[2];

 **any** **from** p.**receive**(MyMessage) -> **@index value** i;

 // checking receiving MyMessage on any port of the port array p and storing the index of the

 // port on which the matching was successful first; if, for example MyMessage is matched first

 // on p[4,2], the content of i will be {4,2}

### 22.2.3 The Trigger operation

The **trigger** operation is used to await a specific message on an incoming port queue.

***Syntactical Structure***

( *Port* | **any** **port** | **any from** PortArrayRef ) "." **trigger**

[ "(" *TemplateInstance* ")" ]

[ **from** *Address* ]

[ "->" [ **value** ( *VariableRef* |

 ( "(" { *VariableRef* [ ":=" [ @decoded [ "(" *Expression* ")" ] ]

 *FieldOrTypeReference* ][","] } ")" )

 ) ]

 [ **sender** *VariableRef* ]

 [ @**index** **value** *VariableRef* ] ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

***Semantic Description***

The **trigger** operation removes the top message from the associated incoming port queue. If that top message meets the matching criteria, the **trigger** operation behaves in the same manner as a **receive** operation. If that top message does not fulfil the matching criteria, it shall be removed from the queue without any further action.

The **trigger** operation requires the port name, matching criteria for type and value, an optional **from** restriction (i.e. selection of communication partner) and an optional assignment of the matching message and sender component to variables.

**Matching criteria**

The matching criteria as defined in clause 22.2.2 apply also to the **trigger** operation.

**Trigger on any message**

A **trigger** operation with no argument list shall trigger on the receipt of any message. Thus, its meaning is identical to the meaning of receive any message.

**Trigger on any port**

To **trigger** on a message at any port, use the **any port** keywords.

**Trigger on any port from a port array**

To trigger on a message at any port from a specific port array, use the **any from** *PortArrayRef*syntax where PortArrayRefshallbe areference to a port array identifier**.** It is also possible to store the index of a port in a single‑dimensional port array at which the operation was successful to a variable of type integer or, in case of multi‑dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching messages, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

If any port in the port array which is checked for matching contains a message that does not match, this message is removed and the containing **alt** statement is re-evalutated, regardless of whether or not other ports in the port array would meet the trigger criteria.

**Stand-alone trigger**

The **trigger** operation can be used as a stand-alone statement in a behaviour description. In this latter case the **trigger** operation is considered to be shorthand for an **alt** statement with two alternatives (one alternative expecting the message and another alternative consuming all other messages and repeating the alt statement, see ETSI ES 201 873‑4 [1]).

**Storing the received message, parts of the received message or the sender**

Rules in clause 22.2.2 shall apply.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

a) The **trigger** operation shall only be used on message-based ports and the type of the value to be received shall be included in the list of incoming types of the port type definition.

b) A message received by *TriggerOnAnyMessage* shall not be assigned to a variable.

c) Type mismatch at storing the received value or parts of the received value and storing the sender shall cause an error.

NOTE 2: Type mismatch may happen if the types in the receive part are not present or not compatible to the types in the assignment part.

d) *AddressRef* for retrieving the sending entity shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **trigger** operation. No *AddressRef* shall contain the special value **null** at the time of the operation.

e) The *PortArrayRef* shall be a reference to a completely initialized port array .

f) The index redirection shall only be used when the operation is used on an any from port array construct.

g) If the index redirection is used for single-dimensional port arrays, the type of the integer variable shall allow storing the highest index of the respective array.

h) If the index redirection is used for multi-dimensional port arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.

i) If a variable referenced in the **value**, **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **trigger** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **trigger** operation.

j) If the **trigger** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause.

***Examples***

EXAMPLE 1: Basic trigger

 MyPort.**trigger**(MyType:?);

 // Specifies that the operation will trigger on the reception of the first message observed of

 // the type MyType with an arbitrary value at port MyPort.

EXAMPLE 2: Trigger from a sender and with storing message or sender

 MyPort.**trigger**(MyType:?) **from** MyPartner;

 // Triggers on the reception of the first message of type MyType at port MyPort

 // received from MyPartner.

 MyPort.**trigger**(MyType:?) **from** MyPartner -> **value** MyRecMessage;

 // This example is almost identical to the previous example. In addition, the message which

 // triggers i.e. all matching criteria are met, is stored in the variable MyRecMessage.

 MyPort.**trigger**(MyType:?) -> **sender** MyPartner;

 // This example is almost identical to the first example. In addition, the reference of the

 // sender component will be retrieved and stored in variable MyPartner.

 MyPort.**trigger**(integer:?) -> **value** MyVar **sender** MyPartner;

 // Trigger on the reception of an arbitrary integer value which afterwards is stored in

 // variable MyVar. The reference of the sender component will be stored in variable MyPartner.

EXAMPLE 3: Trigger on any message

 MyPort.**trigger**;

 MyPort.**trigger** **from** MyPartner;

 MyPort.**trigger** -> **sender** MySenderVar;

EXAMPLE 4: Trigger on any port

 **any port**.**trigger**

EXAMPLE 5: Trigger on any port from port array

 **type** **port** MyPort **message** { **inout** **integer** }

 **type** **component** MyComponent {

 **port** MyPort p[10][10];

 }

 **var** **integer** i[2];

 **any** **from** p.**trigger**(MyMessage) -> @**index** **value** i;

 // Checking if MyMessage has been received on any port of the port array p; if yes, the index

 // of the port on which the matching was first successful is stored in the array i; if no port

 // succeeds, the top messages are removed and the port array is re-checked.

## 22.3 Procedure-based communication

The operations for procedure-based communication via synchronous ports are summarized in table 23.

Table 23: Overview of procedure-based communication

|  |  |
| --- | --- |
| Communication operation | Keyword |
| Invoke procedure call | **call** |
| Accept procedure call from remote entity | **getcall** |
| Reply to procedure call from remote entity | **reply** |
| Raise exception (to an accepted call) | **raise** |
| Handle response from a previous call | **getreply** |
| Catch exception (from called entity) | **catch** |
| Check call/exception/reply received | **check** |

### 22.3.1 The Call operation

The **call** operation specifies the call of a remote operation on another test component or within the SUT.

***Syntactical Structure***

*Port* "." **call** "(" *TemplateInstance* [ "," *CallTimerValue* ]")"

[ **to** *Address* ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**all component**".

***Semantic Description***

The **call** operation is used to specify that a test component calls a procedure in the SUT or in another test component.

The information to be transmitted in the send part of the **call** operation is a signature that may either be defined in the form of a signature template or be defined in-line.

**Handling responses and exceptions to a call**

In case of non-blocking procedure-based communication the handling of exceptions to **call** operations is done by using **catch** (see clause 22.3.6) operations as alternatives in **alt** statements.

If the **nowait** option is used, the handling of responses or exceptions to **call** operations is done by using **getreply** (see clause 22.3.4) and **catch** (see clause 22.3.6) operations as alternatives in **alt** statements.

In case of blocking procedure-based communication, the handling of responses or exceptions to a call is done in the response and exception handling part of the **call** operation by means of **getreply** (see clause 22.3.4) and **catch** (see clause 22.3.6) operations.

The response and exception handling part of a **call** operation looks similar to the body of an **alt** statement. It defines a set of alternatives, describing the possible responses and exceptions to the call.

If necessary, it is possible to enable/disable an alternative by means of a **boolean** expression placed between the "[ ]" brackets of the alternative.

The response and exception handling part of a call operation is executed like an **alt** statement without any active default. This means a corresponding snapshot includes all information necessary to evaluate the (optional) Boolean guards, may include the top element (if any) of the port over which the procedure has been called and may include a timeout exception generated by the (optional) timer that supervises the call.

**Handling timeout exceptions to a call**

The **call** operation may optionally include a timeout. This is defined as an explicit value or constant of **float** type and defines the length of time after the **call** operation has started that a **timeout** exception shall be generated by the test system. If no timeout value part is present in the **call** operation, no **timeout** exception shall be generated.

**Nowait calls of blocking procedures**

Using the keyword **nowait** instead of a timeout exception value in a **call** operation allows calling a procedure to continue without waiting either for a response or an exception raised by the called procedure or a timeout exception.

If the **nowait** keyword is used, a possible response or exception of the called procedure has to be removed from the port queue by using a **getreply** or a **catch** operation in a subsequent **alt** statement.

**Calling blocking procedures without return value, out parameters, inout parameters and exceptions**

A blocking procedure may have no return values, no out and inout parameters and may raise no exception. The call operation for such a procedure shall also have a response and exception handling part to handle the blocking in a uniform manner.

**Calling non-blocking procedures**

A non-blocking procedure has no out and inout parameters, no return value and the non-blocking property is indicated in the corresponding signature definition by means of a **noblock** keyword.

Possible exceptions raised by non-blocking procedures have to be removed from the port queue by using **catch** operations in subsequent **alt** or **interleave** statements.

**Unicast, multicast and broadcast calls of procedures**

Like for the **send** operation, TTCN‑3 also supports unicast, multicast and broadcast calls of procedures. This can be done in the same manner as described in clause 22.2.1, i.e. the argument of the **to** clause of a **call** operation is for unicast calls the address of one receiving entity (or can be omitted in case of one-to-one connections), for multicast calls a list of addresses of a set of receivers and for broadcast calls the **all component** keyword. In case of one-to-one connections, the **to** clause may be omitted, because the receiving entity is uniquely identified by the system structure.

The handling of responses and exceptions for a blocking or non-blocking unicast **call** operation has been explained in this clause under "Handling timeout exceptions to a call". A multicast or broadcast **call** operation may cause several responses and exceptions from different communication partners.

In case of a multicast or broadcast **call** operation of a non-blocking procedure, all exceptions which may be raised from the different communication partners can be handled in subsequent **catch**, **alt** or **interleave** statements.

In case of a multicast or broadcast **call** operation of a blocking procedure, two options exist. Either, only one response or exception is handled in the response and exception handling part of the **call** operation. Then, further responses and exceptions can be handled in subsequent **alt** or **interleave** statements. Or, several responses or exceptions are handled by the use of repeat statements in one or more of the statement blocks of the response and exception handling part of the call operation: the execution of a repeat statement causes the re-evaluation of the call body.

NOTE 2: In the second case, the user needs to handle the number of repetitions.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

a) The **call** operation shall only be used on procedure-based ports. The type definition of the port at which the call operation takes place shall include the procedure name in its **out** or **inout** list i.e. it shall be allowed to call this procedure at this port.

b) All **in** and **inout** parameters of the signature shall have a specific value i.e. the use of matching mechanisms such as *AnyValue* is not allowed.

c) Only out parameters may be omitted or specified with a matching attribute.

d) The signature arguments of the **call** operation are not used to retrieve variable names for **out** and **inout** parameters. The actual assignment of the procedure return value and **out** and **inout** parameter values to variables shall explicitly be made in the response and exception handling part of the **call** operation by means of **getreply** and **catch** operations. This allows the use of signature templates in **call** operations in the same manner as templates can be used for types.

e) A **to** clause shall be present in case of one-to-many connections.

f) *AddressRef* shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **call** operation. No *AddressRef* shall contain the special value **null** at the time of the operation.

g) *CallTimerValue* shall be of type float.

h) The selection of the alternatives to a call shall only be based on **getreply** and **catch** operations for the called procedure. Unqualified **getreply** and **catch** operations shall only treat replies from and exceptions raised by the called procedure. The use of **else** branches and the invocation of altsteps is not allowed.

i) The evaluation of the Boolean expressions guarding the alternatives in the response and exception handling part may have side effects. In order to avoid unexpected side effects, the same rules as for the Boolean guards in **alt** statements shall be applied (see clause 20.2).

j) The call operation for a blocking procedures without return value, out parameters, inout parameters and exceptions shall also have a response and exception handling part to handle the blocking in a uniform manner.

k) In case of a multicast or broadcast **call** operation of a blocking procedure, where the **nowait** keyword is used, all responses and exceptions have to be handled in subsequent **alt** or **interleave** statements.

l) The **call** operation for a non-blocking procedure shall have no response and exception handling part, shall raise no timeout exception and shall not use the **nowait** keyword.

m) Applying a **call** operation to an unmapped or disconnected port shall cause a test case error.

***Examples***

EXAMPLE 1: Blocking call with getreply

 // Given …

 **signature** MyProc (**out** integer MyPar1, **inout** boolean MyPar2);

 :

 // a call of MyProc

 MyPort.**call**(MyProc:{ -, MyVar2}) { // in-line signature template for the call of MyProc

 [] MyPort.**getreply**(MyProc:{?, ?}) { }

 }

 // … and another call of MyProc

 MyPort.**call**(MyProcTemplate) { // using signature template for the call of MyProc

 [] MyPort.**getreply**(MyProc:{?, ?}) { }

 }

 MyPort.**call**(MyProcTemplate) **to** MyPeer { // calling MyProc at MyPeer

 [] MyPort.**getreply**(MyProc:{?, ?}) { }

 }

EXAMPLE 2: Blocking call with getreply and catch

 // Given

 **signature** MyProc3 (**out** **integer** MyPar1, **inout** **boolean** MyPar2) **return** MyResultType

 **exception** (ExceptionTypeOne, ExceptionTypeTwo);

 :

 // Call of MyProc3

 MyPort.**call**(MyProc3:{ -, **true** }) **to** MyPartner {

 [] MyPort.**getreply**(MyProc3:{?, ?}) -> **value** MyResult **param** (MyPar1Var,MyPar2Var) { }

 [] MyPort.**catch**(MyProc3, MyExceptionOne) {

 **setverdict**(**fail**);

 **stop**;

 }

 [] MyPort.**catch**(MyProc3, ExceptionTypeTwo : ?) {

 **setverdict**(**inconc**);

 }

 [MyCondition] MyPort.**catch**(MyProc3, MyExceptionThree) { }

 }

EXAMPLE 3: Blocking call with timeout exception

 MyPort.call(MyProc:{5,MyVar}, 20E-3) {

 [] MyPort.**getreply**(MyProc:{?, ?}) { }

 [] MyPort.**catch**(**timeout**) { // timeout exception after 20ms

 **setverdict**(**fail**);

 **stop**;

 }

 }

EXAMPLE 4: Nowait call

 MyPort.**call**(MyProc:{5, MyVar}, **nowait**); // The calling test component will continue

 // its execution without waiting for the

 // termination of MyProc

EXAMPLE 5: Blocking call without return value, out parameters, inout parameters and exceptions

 // Given …

 **signature** MyBlockingProc (**in** integer MyPar1, **in** boolean MyPar2);

 :

 // a call of MyBlockingProc

 MyPort.**call**(MyBlockingProc:{ 7, **false** }) {

 [] MyPort.**getreply**( MyBlockingProc:{ -, - } ) { }

 }

EXAMPLE 6: Broadcast call

 **var** **boolean** first:= **true**;

 MyPort.**call**(MyProc:{5,MyVar}, 20E-3) **to all** **component** { // Broadcast call of MyProc

 // Handles the response from MyPeerOne

 [first] MyPort.**getreply**(MyProc:{?, ?}) **from** MyPeerOne {

 **if** (first) { first := **false**; **repeat**; }

 :

 }

 // Handles the response from MyPeerTwo

 [first] MyPort.**getreply**(MyProc:{?, ?}) **from** MyPeerTwo {

 **if** (first) { first := **false**; **repeat**; }

 :

 }

 [] MyPort.**catch**(**timeout**) { // timeout exception after 20ms

 **setverdict**(**fail**);

 **stop**;

 }

 }

 **alt** {

 [] MyPort.**getreply**(MyProc:{?, ?}) { // Handles all other responses to the broadcast call

 **repeat**

 }

 }

EXAMPLE 7: Multicast call

 MyPort.**call**(MyProc:{5,MyVar}, nowait) **to** (MyPeer1, MyPeer2); // Multicast call of MyProc

 **interleave** {

 [] MyPort.**getreply**(MyProc:{?, ?}) **from** MyPeer1 { } // Handles the response of MyPeer1

 [] MyPort.**getreply**(MyProc:{?, ?}) **from** MyPeer2 { } // Handles the response of MyPeer2

 }

### 22.3.2 The Getcall operation

The **getcall** operation is used to accept calls.

***Syntactical Structure***

( *Port* | **any** **port** | **any from** PortArrayRef ) "." **getcall**

[ "(" *TemplateInstance* ")" ]

[ **from** *Address* ]

[ "->" [ **param** "(" { ( *VariableRef* ":=" [ @decoded [ "(" *Expression* ")" ] ] *ParameterIdentifier* ) "," } *|*

 { ( *VariableRef* | "-" ) "," }

 ")" ]

 [ **sender** *VariableRef* ]

 [ @**index** **value** *VariableRef* ] ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

***Semantic Description***

The **getcall** operation is used to specify that a test component accepts a call from the SUT, or another test component.

The **getcall** operation shall remove the top call from the incoming port queue, if, and only if, the matching criteria associated to the **getcall** operation are fulfilled. These matching criteria are related to the signature of the call to be processed and the communication partner. The matching criteria for the signature may either be specified in-line or be derived from a signature template.

The assignment of **in** and **inout** parameter values to variables shall be made in the assignment part of the **getcall** operation. This allows the use of signature templates in **getcall** operations in the same manner as templates are used for types.

A **getcall** operation may be restricted to a certain communication partner in case of one-to-many connections. This restriction shall be denoted by using the **from** keyword.

The (optional) assignment part of the **getcall** operation comprises the assignment of **in** and **inout** parameter values to variables and the retrieval of the address of the calling component. The keyword **param** is used to retrieve the parameter values of a call.

When assigning individual parameters of a call, encoded parameters can be decoded prior to assignment using the **@decoded** modifier. In this case, the referenced parameter on the right hand sided of the assignment shall be one of the **bitstring**, **hexstring**, **octetstring**, **charstring** or **universal** **charstring** types. It shall be decoded into a value of the same type as the variable on the left hand side of the assignment. Failure of this decoding shall cause a test case error. In case the referenced field is of the **universal** **charstring** type, the **@decoded** clause can contain an optional parameter defining the encoding format. The parameter shall be of the **charstring** type and it shall contain one of the strings allowed for the **decvalue\_unichar** function (specified in clause C.5.4). Any other value shall cause an error. In case the referenced field is not a **universal** **charstring**, the optional parameter shall not be present.

The keyword **sender** is used when it is required to retrieve the address of the sender (e.g. for addressing a **reply** or exception to the calling party in a one-to-many configuration).

**Accepting any call**

A **getcall** operation with no argument list for the signature matching criteria will remove the call on the top of the incoming port queue (if any) if all other matching criteria are fulfilled.

**Getcall on any port**

To **getcall** on any port is denoted by the **any** keyword.

**Getcall on any port from a port array**

To **getcall** on any port from a specific port array, use the **any from** *PortArrayRef*syntax where PortArrayRefshallbe areference to a port array identifier**.** It is also possible to store the index of a port in a single-dimensional port array at which the operation was successful to a variable of type integer or, in case of multi‑dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching calls, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

a) The **getcall** operation shall only be used on procedure-based ports and the signature of the procedure call to be accepted shall be included in the list of allowed incoming procedures of the port type definition.

b) The signature argument of the **getcall** operation shall not be used to pass in variable names for **in** and **inout** parameters.

c) The *ParameterIdentifier*s shall be from the corresponding signature definition.

d) The value assignment part shall not be used with the getcall operation.

e) Parameters of calls accepted by *accepting any call* shall not be assigned to a variable, i.e. the **param** clause shall not be present.

f) *AddressRef* for retrieving the sending entity shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **getcall** operation. No *AddressRef* shall contain the special value **null** at the time of the operation.

g) The *PortArrayRef* shall be a reference to a completely initialized port array.

h) The index redirection shall only be used when the operation is used on an any from port array construct.

i) If the index redirection is used for single-dimensional port arrays, the type of the integer variable shall allow storing the highest index of the respective array.

j) If the index redirection is used for multi-dimensional port arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective array, and its type shall allow storing the highest index (from all dimensions) of the array.

k) If a variable referenced in the **param**, **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **getcall** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **getcall** operation.

l) If the **getcall** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause.

NOTE 2: An error due to a type mismatch may happen if the types in the receive part are not present or not compatible to the types in the assignment part.

m) When assigning implicitly decoded parameters (by using the **@decoded** modifier) in cases where the value or template to be matched uses the *MatchDecodedContent* (**decmatch**) matching for the parameter to be stored, the type of the template in the *MatchDecodedContent* matching shall be type-compatible to the type of the variable the decoded field is stored into.

***Examples***

EXAMPLE 1: Basic getcall

 MyPort.**getcall**(MyProc: MyProcTemplate(5, MyVar)); // accepts a call of MyProc at MyPort

 MyPort.**getcall**(MyProc:{5, MyVar}) **from** MyPeer; // accepts a call of MyProc at MyPort from MyPeer

EXAMPLE 2: Getcall with matching and assignments of parameter values to variables

 MyPort.**getcall**(MyProc:{?, ?}) **from** MyPartner -> **param** (MyPar1Var, MyPar2Var);

 // The in or inout parameter values of MyProc are assigned to MyPar1Var and MyPar2Var.

 MyPort.**getcall**(MyProc:{5, MyVar}) -> **sender** MySenderVar;

 // Accepts a call of MyProc at MyPort with the in or inout parameters 5 and MyVar.

 // The address of the calling party is retrieved and stored in MySenderVar.

 // The following getcall examples show the possibilities to use matching attributes

 // and omit optional parts, which may be of no importance for the test specification.

 MyPort.**getcall**(MyProc:{5, MyVar}) -> **param**(MyVar1, MyVar2) **sender** MySenderVar;

 MyPort.**getcall**(MyProc:{5, ?}) -> **param**(MyVar1, MyVar2);

 MyPort.**getcall**(MyProc:{?, MyVar}) -> **param**( - , MyVar2);

 // The value of the first inout parameter is not important or not used

 // The following examples shall explain the possibilities to assign in and inout parameter

 // values to variables. The following signature is assumed for the procedure to be called:

 **signature** MyProc2(**in** **integer** A, **integer** B, **integer** C, **out** **integer** D, **inout** **integer** E);

 MyPort.**getcall**(MyProc2:{?, ?, 3, - , ?}) -> **param** (MyVarA, MyVarB, - , -, MyVarE);

 // The parameters A, B, and E are assigned to the variables MyVarA, MyVarB, and

 // MyVarE. The out parameter D needs not to be considered.

 MyPort.**getcall**(MyProc2:{?, ?, 3, -, ?}) -> **param** (MyVarA:= A, MyVarB:= B, MyVarE:= E);

 // Alternative notation for the value assignment of in and inout parameter to variables. Note,

 // the names in the assignment list refer to the names used in the signature of MyProc2

 MyPort.**getcall**(MyProc2:{1, 2, 3, -, \*}) -> **param** (MyVarE:= E);

 // Only the inout parameter value is needed for the further test case execution

 // The following example demonstrates the use of encoded parameters:

 **signature** MyProc3(**in** **integer** paramType, **octetstring** encodedParam);

 **template integer** m\_int := ?;

 …

 **var integer** v\_myVarX;

 MyPort.**getcall**(MyProc3:{1, **decmatch** m\_int}) -> **param** (v\_myVarX := **@decoded** encodedParam);

 // The parameters encodedParam is decoded into an integer and assigned to v\_myVarX.

EXAMPLE 3: Accepting any call

 MyPort.**getcall**; // Removes the top call from MyPort.

 MyPort.**getcall** **from** MyPartner; // Removes a call from MyPartner from port MyPort

 MyPort.**getcall** -> **sender** MySenderVar; // Removes a call from MyPort and retrieves

 // the address of the calling entity

EXAMPLE 4: Getcall on any port

 **any** **port**.**getcall**(MyProc:?)

EXAMPLE 5: Getcall on any port from port array

 **type** **port** MyPort **procedure** { **inout** MyProc }

 **type** **component** MyComponent {

 **port** **MyPort** p[10][10];

 }

 **var** **integer** i[2];

 **any** **from** p.**getcall**(MyProc:?) -> @**index** **value** i;

 // checking for an incoming call of the type MyProc on any port of the port array p and storing

 // the index of the port on which the matching was successful first

### 22.3.3 The Reply operation

The **reply** operation is used to reply to a call.

***Syntactical Structure***

*Port* "." **reply** "(" *TemplateInstance* [ **value** *Expression* ] ")"

[ **to** *Address* ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**all component**".

***Semantic Description***

The **reply** operation is used to reply to a previously accepted call according to the procedure signature.

NOTE 2: The relation between an accepted call and a **reply** operation cannot always be checked statically. For testing it is allowed to specify a **reply** operation without an associated **getcall** operation.

The value part of the **reply** operation consists of a signature reference with an associated actual parameter list and (optional) return value. The signature may either be defined in the form of a signature template or it may be defined in‑line.

Responses to one or more **call** operations may be sent to one, several or all peer entities connected to the addressed port. This can be specified in the same manner as described in clause 22.2.1. This means, the argument of the **to** clause of a **reply** operation is for unicast responses the address of one receiving entity, for multicast responses a list of addresses of a set of receivers and for broadcast responses the **all component** keywords.

In case of one-to-one connections, the **to** clause may be omitted, because the receiving entity is uniquely identified by the system structure.

A return value shall be explicitly stated with the **value** keyword.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

a) A **reply** operation shall only be used at a procedure-based port. The type definition of the port shall include the name of the procedure to which the **reply** operation belongs.

b) All **out** and **inout** parameters of the signature shall have a specific value i.e. the use of matching mechanisms such as *AnyValue* is not allowed.

c) A **to** clause shall be present in case of one-to-many connections.

d) *AddressRef* shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **reply** operation. No *AddressRef* shall contain the special value **null** at the time of the operation.

e) If a value is to be returned to the calling party, this shall be explicitly stated using the **value** keyword.

f) Applying a **reply** operation to an unmapped or disconnected port shall cause a test case error.

***Examples***

 MyPort.**reply**(MyProc2:{ - ,5}); // Replies to an accepted call of MyProc2.

 MyPort.**reply**(MyProc2:{ - ,5}) **to** MyPeer; // Replies to an accepted call of MyProc2 from MyPeer

 MyPort.**reply**(MyProc2:{ - ,5}) **to** (MyPeer1, MyPeer2); // Multicast reply to MyPeer1 and MyPeer2

 MyPort.**reply**(MyProc2:{ - ,5}) **to** **all component**; // Broadcast reply to all entities connected

 // to MyPort

 MyPort.**reply**(MyProc3:{5,MyVar} **value** 20); // Replies to an accepted call of MyProc3.

### 22.3.4 The Getreply operation

The **getreply** operation is used to handle replies from a previously called procedure.

***Syntactical Structure***

( *Port* | **any** **port** | **any from** PortArrayRef ) "." **getreply**

[ "(" *TemplateInstance* [ **value** *TemplateInstance* ]")" ]

[ **from** *Address* ]

[ "->" [ **value** (*VariableRef* |

 ( "(" { *VariableRef* [ ":=" [ @decoded [ "(" *Expression* ")" ] ]
 *FieldOrTypeReference* ][","] } ")" )

 )]

 [ **param** "(" { ( *VariableRef* ":=" [ @decoded [ "(" *Expression* ")" ] ]
 *ParameterIdentifier* ) "," } *|*

 { ( *VariableRef* | "-" ) "," }

 ")" ]

 [ **sender** *VariableRef* ]

 [ @**index** **value** *VariableRef* ] ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

***Semantic Description***

The **getreply** operation is used to handle replies from a previously called procedure.

The **getreply** operation shall remove the top reply from the incoming port queue, if, and only if, the matching criteria associated to the **getreply** operation are fulfilled. These matching criteria are related to the signature of the procedure to be processed and the communication partner. The matching criteria for the signature may either be specified in-line or be derived from a signature template.

Matching against a received return value can be specified by using the **value** keyword.

A **getreply** operation may be restricted to a certain communication partner in case of one-to-many connections. This restriction shall be denoted by using the **from** keyword.

The assignment of **out** and **inout** parameter values to variables shall be made in the assignment part of the **getreply** operation. This allows the use of signature templates in **getreply** operations in the same manner as templates are used for types.

The (optional) assignment part of the **getreply** operation comprises the assignment of **out** and **inout** parameter values to variables and the retrieval of the address of the sender of the reply. The keyword **value** is used to retrieve return values and the keyword **param** is used to retrieve the parameter values of a reply. The keyword **sender** is used when it is required to retrieve the address of the sender.

When assigning individual parameters of a reply, encoded parameters can be decoded prior to assignment using the **@decoded** modifier. In this case, the referenced parameter on the right hand sided of the assignment shall be one of the **bitstring**, **hexstring**, **octetstring**, **charstring** or **universal** **charstring** types. It shall be decoded into a value of the same type as the variable on the left hand side of the assignment. Failure of this decoding shall cause a test case error. In case the referenced field is of the **universal** **charstring** type, the **@decoded** clause can contain an optional parameter defining the encoding format. The parameter shall be of the **charstring** type and it shall contain one of the strings allowed for the **decvalue\_unichar** function (specified in clause C.5.4). Any other value shall cause an error. In case the referenced field is not a **universal** **char string**, the optional parameter shall not be present.

**Get any reply**

A **getreply** operation with no argument list for the signature matching criteria shall remove the reply message on the top of the incoming port queue (if any) if all other matching criteria are fulfilled.

If *GetAnyReply* is used in the response and exception handling part of a **call** operation, it shall only treat replies from the procedure invoked by the **call** operation.

**Get a reply on any port**

To get a reply on any port, use the **any port** keywords.

**Get a reply on any port from a port array**

To get a reply on any port from a specific port array, use the **any from** *PortArrayRef*syntax where PortArrayRefshallbe areference to a port array identifier**.** It is also possible to store the index of a port in a single‑dimensional port array at which the operation was successful to a variable of type integer or, in case of multi‑dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching replies, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

a) A **getreply** operation shall only be used at a procedure-based port. The type definition of the port shall include the name of the procedure to which the **getreply** operation belongs.

b) The signature argument of the **getreply** operation shall not be used to pass in variable names for **out** and **inout** parameters.

c) Parameters or return values of responses accepted by *get any reply* shall not be assigned to a variable, i.e. the **param** and **value** clause shall not be present.

d) *AddressRef* for retrieving the sending entity shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **getreply** operation. No *AddressRef* shall contain the special value **null** at the time of the operation.

e) The *PortArrayRef* shall be a reference to a completely initialized port array .

f) The index redirection shall only be used when the operation is used on an any from port array construct.

g) If the index redirection is used for single-dimensional arrays, the type of the integer variable shall allow storing the highest index of the respective port array.

h) If the index redirection is used for multi-dimensional arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective port array, and the its type shall allow storing the highest index (from all dimensions) of the port array.

i) If a variable referenced in the **value**, **param**, **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **getreply** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **getreply** operation.

j) If the **getreply** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause.

NOTE 2: An error due to a type mismatch may happen if the types in the receive part are not present or not compatible to the types in the assignment part.

k) When assigning implicitly decoded parameters (by using the **@decoded** modifier) in cases where the value or template to be matched uses the *MatchDecodedContent* (**decmatch**) matching for the parameter to be stored, the type of the template in the *MatchDecodedContent* matching shall be type-compatible to the type of the variable the decoded field is stored into.

***Examples***

EXAMPLE 1: Basic getreply

 MyPort.**getreply**(MyProc:{5, ?} **value** 20); // Accepts a reply of MyProc with two out or

 // inout parameters and a return value of 20

 MyPort.**getreply**(MyProc2:{ - ,5}) **from** MyPeer; // Accepts a reply of MyProc2 from MyPeer

EXAMPLE 2: Getreply with storing inout/out parameters and return values in variables

 MyPort.**getreply**(MyProc1:{?, ?} **value** ?) -> **value** MyRetValue **param**(MyPar1,MyPar2);

 // The returned value is assigned to variable MyRetValue and the value

 // of the two out or inout parameters are assigned to the variables MyPar1 and MyPar2.

 MyPort.**getreply**(MyProc1:{?, ?} **value** ?) -> **value** MyRetValue **param**( - , MyPar2) **sender** MySender;

 // The value of the first parameter is not considered for the further test execution and

 // the address of the sender component is retrieved and stored in the variable MySender.

 // The following examples describe some possibilities to assign out and inout parameter values

 // to variables. The following signature is assumed for the procedure which has been called

 **signature** MyProc2(**in** **integer** A, **integer** B, **integer** C, **out** **integer** D, **inout** **integer** E);

 MyPort.**getreply**(ATemplate) -> **param**( - , - , - , MyVarOut1, MyVarInout1);

 MyPort.**getreply**(ATemplate) -> **param**(MyVarOut1:=D, MyVarOut2:=E);

 MyPort.**getreply**(MyProc2:{ - , - , - , 3, ?}) -> **param**(MyVarInout1:=E);

 // The following example demonstrates the use of encoded parameters:

 **signature** MyProc3(**out** **integer** paramType, **out** **octetstring** encodedParam);

 **template integer** m\_int := ?;

 …

 **var integer** v\_myVarX;

 MyPort.**getreply**(MyProc3:{1, **decmatch** m\_int}) -> **param** (v\_myVarX := **@decoded** encodedParam);

 // The parameters encodedParam is decoded into an integer and assigned to v\_myVarX.

EXAMPLE 3: Get any reply

 MyPort.**getreply**; // Removes the top reply from MyPort.

 MyPort.**getreply** **from** MyPeer; // Removes the top reply received from MyPeer from MyPort.

 MyPort.**getreply** -> **sender** MySenderVar; // Removes the top reply from MyPort and retrieves the

 // address of the sender entity

EXAMPLE 4: Get a reply on any port

 **any** **port**.**getreply**(Myproc:?)

EXAMPLE 5: Get a reply on any port from port array

 **type** **port** MyPort **procedure** { **inout** MyProc }

 **type** **component** MyComponent {

 **port** MyPort p[10][10];

 }

 **var** **integer** i[2];

 **any** **from** p.**getreply**(MyProc:?) -> @**index** **value** i;

 // Getting a reply of the type MyProc on any port of the port array p and

 // storing the index of the port on which the matching was successful first

### 22.3.5 The Raise operation

Exceptions are raised with the **raise** operation.

***Syntactical Structure***

*Port* "." **raise** "(" *Signature* "," *TemplateInstance* ")"

[ **to** *Address* ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**all component**".

***Semantic Description***

The **raise** operation is used to raise an exception.

NOTE 2: The relation between an accepted call and a **raise** operation cannot always be checked statically. For testing it is allowed to specify a **raise** operation without an associated **getcall** operation.

The value part of the **raise** operation consists of the signature reference followed by the exception value.

Exceptions are specified as types. Therefore the exception value may either be derived from a template or be the value resulting from an expression (which of course can be an explicit value). The optional type field in the value specification to the **raise** operation shall be used in cases where it is necessary to avoid any ambiguity of the type of the value being sent.

Exceptions to one or more **call** operations may be sent to one, several or all peer entities connected to the addressed port. This can be specified in the same manner as described in clause 22.2.1. This means, the argument of the **to** clause of a **raise** operation is for unicast exceptions the address of one receiving entity, for multicast exceptions a list of addresses of a set of receivers and for broadcast exceptions the **all component** keywords.

In case of one-to-one connections, the **to** clause may be omitted, because the receiving entity is uniquely identified by the system structure.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

a) An exception shall only be raised at a procedure-based port. An exception is a reaction to an accepted procedure call the result of which leads to an exceptional event.

b) The type of the exception shall be specified in the signature of the called procedure. The type definition of the port shall include in its list of accepted procedure calls the name of the procedure to which the exception belongs.

c) A **to** clause shall be present in case of one-to-many connections.

d) *AddressRef* shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **raise** operation. No *AddressRef* shall contain the special value **null** at the time of the operation.

e) Applying a **raise** operation to an unmapped or disconnected port shall cause a test case error.

***Examples***

 MyPort.**raise**(MySignature, MyVariable + YourVariable - 2);

 // Raises an exception with a value which is the result of the arithmetic expression

 // at MyPort

 MyPort.**raise**(MyProc, **integer**:5}); // Raises an exception with the integer value 5 for MyProc

 MyPort**.raise**(MySignature, "My string") **to** MyPartner;

 // Raises an exception with the value "My string" at MyPort for MySignature and

 // send it to MyPartner

 MyPort**.raise**(MySignature, "My string") **to** (MyPartnerOne, MyPartnerTwo);

 // Raises an exception with the value "My string" at MyPort and sends it to MyPartnerOne and

 // MyPartnerTwo (i.e. multicast communication)

 MyPort**.raise**(MySignature, "My string") **to** **all component**;

 // Raises an exception with the value "My string" at MyPort for MySignature and sends it

 // to all entites connected to MyPort (i.e. broadcast communication)

### 22.3.6 The Catch operation

The **catch** operation is used to catch exceptions.

***Syntactical Structure***

( *Port* | **any** **port** | **any from** PortArrayRef ) "." **catch**

[ "(" ( *Signature* "," *TemplateInstance* ) | *TimeoutKeyword* ")" ]

[ **from** *Address* ]

[ "->" [ **value** ( *VariableRef* |

 ( "(" { *VariableRef* [ ":=" [ @decoded [ "(" *Expression* ")" ] ] *FieldOrTypeReference* ][","] } ")" )

 ) ]

 [ **sender** *VariableRef* ]

 [ @**index** **value** *VariableRef* ] ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

***Semantic Description***

The **catch** operation is used to catch exceptions raised by a test component or the SUT as a reaction to a procedure call. Exceptions are specified as types and thus, can be treated like messages, e.g. templates can be used to distinguish between different values of the same exception type.

The **catch** operation removes the top exception from the associated incoming port queue if, and only if, that top exception satisfies all the matching criteria associated with the **catch** operation.

A **catch** operation may be restricted to a certain communication partner in case of one-to-many connections. This restriction shall be denoted by using the **from** keyword.

The (optional) redirection part of the **catch** operation comprises of storing the exception value and/or one or more parts of it and the retrieval of the address of the calling component. The keyword **value** is used to retrieve the value of an exception and/or the parts of it and the keyword **sender** is used when it is required to retrieve the address of the sender.

When assigning individual fields of an exception, encoded payload fields can be decoded prior to assignment using the **@decoded** modifier. In this case, the referenced field on the right hand sided of the assignment shall be one of the **bitstring**, **hexstring**, **octetstring**, **charstring** or **universal** **charstring** types. It shall be decoded into a value of the same type as the variable on the left hand side of the assignment. Failure of this decoding shall cause a test case error. In case the referenced field is of the **universal** **charstring** type, the **@decoded** clause can contain an optional parameter defining the encoding format. The parameter shall be of the **charstring** type and it shall contain one of the strings allowed for the **decvalue\_unichar** function (specified in clause C.5.4). Any other value shall cause an error. In case the referenced field is not a **universal** **charstring**, the optional parameter shall not be present.

The **catch** operation may be part of the response and exception handling part of a **call** operation or be used to determine an alternative in an **alt** statement. If the **catch** operation is used in the accepting part of a **call** operation, the information about port name and signature reference to indicate the procedure that raised the exception is redundant, because this information follows from the **call** operation. However, for readability reasons (e.g. in case of complex **call** statements) this information shall be repeated.

**The Timeout exception**

There is one special **timeout** exception that can be caught by the **catch** operation. The **timeout** exception is an emergency exit for cases where a called procedure neither replies nor raises an exception within a predetermined time (see clause 22.3.1).

**Catch any exception**

A **catch** operation with no argument list allows any valid exception to be caught. The most general case is without using the **from** keyword. *CatchAnyException* will also catch the **timeout** exception.

**Catch on any port**

To **catch** an exception on any port use the **any** keyword.

**Catch on any port from a port array**

To **catch** an exception on any port from a specific port array, indices use the **any from** *PortArrayRef*syntax where PortArrayRefshallbe areference to a port array identifier**.** It is also possible to store the index of a port in a single-dimensional port array at which the operation was successful to a variable of type integer or, in case of multi‑dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for matching exceptions, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

The catch on any port from a port array operation can not be used to catch a call timeout.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

1. The **catch** operation shall only be used at procedure-based ports. The type of the caught exception shall be specified in the signature of the procedure that raised the exception.
2. No binding of the incoming values to the terms of the expression or to the template shall occur. The assignment of the exception values to variables shall be made in the assignment part of the **catch** operation.
3. Catching **timeout** exceptions shall be restricted to the exception handling part of a call. No further matching criteria (including a **from** part) and no assignment part is allowed for a **catch** operation that handles a **timeout** exception.
4. Exception values accepted by *catch any exception* shall not be assigned to a variable, i.e. the **value** clause shall not be present.
5. If *CatchAnyException* is used in the response and exception handling part of a **call** operation, it shall only treat exceptions raised by the procedure invoked by the **call** operation.
6. *AddressRef* for retrieving the sending entity shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **catch** operation. No *AddressRef* shall contain the special value **null** at the time of the operation.
7. The *PortArrayRef* shall be a reference to a completely initialized port array.
8. The index redirection shall only be used when the operation is used on an any from port array construct.
9. If the index redirection is used for single-dimensional arrays, the type of the integer variable shall allow storing the highest index of the respective port array.
10. If the index redirection is used for multi-dimensional arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective port array, and the its type shall allow storing the highest index (from all dimensions) of the port array.
11. If a variable referenced in the **value**, **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **catch** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **catch** operation.
12. If the **catch** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause.

NOTE 2: An error due to a type mismatch may happen if the types in the receive part are not present or not compatible to the types in the assignment part.

1. When assigning implicitly decoded exception fields (by using the **@decoded** modifier) in cases where the value or template to be matched uses the *MatchDecodedContent* (**decmatch**) matching for the parameter to be stored, the type of the template in the *MatchDecodedContent* matching shall be type-compatible to the type of the variable the decoded field is stored into.

***Examples***

EXAMPLE 1: Basic catch

 MyPort.**catch**(MyProc, **integer:** MyVar); // Catches an integer exception of value

 // MyVar raised by MyProc at port MyPort.

 MyPort.**catch**(MyProc, MyVar); // Is an alternative to the previous example.

 MyPort.**catch**(MyProc, A<B); // Catches a boolean exception

 MyPort.**catch**(MyProc, MyType:{5, MyVar}); // In-line template definition of an exception value.

 MyPort.**catch**(MyProc, **charstring**:"Hello")**from** MyPeer; // Catches "Hello" exception from MyPeer

EXAMPLE 2: Catch with storing value and/or sender in variables

 MyPort.**catch**(MyProc, MyType:?) **from** MyPartner -> **value** MyVar;

 // Catches an exception from MyPartner and assigns its value to MyVar.

 MyPort.**catch**(MyProc, MyTemplate(5)) -> **value** MyVarTwo **sender** MyPeer;

 // Catches an exception, assigns its value to MyVarTwo and retrieves the

 // address of the sender.

 MyPort.**catch**(MyProc, MyTemplate(5)) -> **value** (MyVarThree:= f1)

 **sender** MyPeer;

 // Catches an exception, assigns the value of its field f1 to MyVarThree and retrieves the

 // address of the sender.

 // Handling encoded exception payload:

 **type** MyException **record** {

 **...**

 }

 **type** CommonException **record** {

 **integer** exceptionId,

 **octetstring** payload

 }

 **signature** S() **exception** (CommonException);

 ...

 **var** MyException v\_myVar;

 MyPort.**catch** (S, CommonException:{exceptionId := 25, payload := **decmatch** MyException:? }) -> **value** (v\_myVar := **@decoded** payload);

 // The encoded payload field of the caught exception is decoded and matched with m\_excTemplate;

 // if the matching is successful the decoded payload is stored in v\_myVar.

EXAMPLE 3: The Timeout exception

 MyPort.**call**(MyProc:{5,MyVar}, 20E-3) {

 [] MyPort.**getreply**(MyProc:{?, ?}) { }

 [] MyPort.**catch**(**timeout**) { // timeout exception after 20ms

 **setverdict**(**fail**);

 **stop**;

 }

 }

EXAMPLE 4: Catch any exception

 MyPort.**catch**;

 MyPort.**catch** **from** MyPartner;

 MyPort.**catch** -> **sender** MySenderVar;

EXAMPLE 5: Catch on any port

 **any port**.**catch;**

EXAMPLE 6: Catch on any port from port array

 **type** **port** MyPort **procedure** { **inout** MyProc }

 **type** **component** MyComponent {

 **port** MyPort p[10][10];

 }

 **var** **integer** i[2];

 **any** **from** p.**catch**(MyProc, MyType:?) -> @**index** **value** i;

 // Catching an incoming exception of type MyType on any port in the port array p and

 // storing the index of the port on which the matching was successful first

## 22.4 The Check operation

The **check** operation allows reading the top element of a message‑based or procedure‑based *incoming* port queue.

***Syntactical Structure***

( *Port* | **any** **port** | **any from** PortArrayRef ) "." **check**

[ "("

 ( *PortReceiveOp | PortGetCallOp | PortGetReplyOp | PortCatchOp* ) |

 ( [ **from** *Address* ]

 [ "->" [ **sender** *VariableRef* ]

 [ **@index** **value** VariableRef ] ] )

 ")" ]

NOTE 1: *Address* may be an *AddressRef*, a list of *AddressRef*-s or "**any component**".

***Semantic Description***

The **check** operation is a generic operation that allows read access to the top element of message‑based and procedure‑based *incoming* port queues without removing the top element from the queue. The **check** operation has to handle values of a certain type at message-based ports and to distinguish between calls to be accepted, exceptions to be caught and replies from previous calls at procedure-based ports.

The receiving operations **receive**, **getcall**, **getreply** and **catch** together with their matching and value, sender or parameter storing parts, are used by the **check** operation to define the conditions that have to be checked and the information to be optionally extracted.

It is the *top* element of an incoming port queue that shall be checked (it is not possible to look *into* the queue). If the queue is empty the **check** operation fails. If the queue is not empty, a copy of the top element is taken and the receiving operation specified in the **check** operation is performed on the copy. The **check** operation fails if the receiving operation fails i.e. the matching criteria are not fulfilled. In this case the *copy* of the top element of the queue is discarded and test execution continues in the normal manner, i.e. the statement or alternative next to the check operation is evaluated. The **check** operation is successful if the receiving operation is successful. In this case, the value, sender or parameter storing parts of the receiving operation, if any, are executed, i.e. the message and/or a part of it, the sender's address or component reference, the parameter(s) of the call or reply or the value of the exception are stored in the associated variables.

If **check** is used as a stand-alone statement, it is considered to be a shorthand for an **alt** statement with the **check** operation as the only alternative.

**Check any operation**

A **check** operation with no argument list allows checking whether something waits for processing in an incoming port queue. The **check** any operation allows to distinguish between different senders (in case of one-to-many connections) by using a **from** clause and to retrieve the sender by using a shorthand assignment part with a **sender** clause.

**Check on any port**

To **check** on any port, use the **any port** keywords.

**Check on any port from a port array**

To **check** on any port from a specific port array, indicesindices use the **any from** *PortArrayRef*syntax where PortArrayRefshallbe areference to a port array identifier**.** It is also possible to store the index of a port in a single‑dimensional port array at which the operation was successful to a variable of type integer or, in case of multi‑dimensional port arrays the index of the successful port to an integer array or record of integer variable. When checking the port array for a matching message, call, reply or exception, the port indices to be checked are iterated from lowest to highest. If the port array is multi-dimensional, then the ports are iterated over from innermost to outermost array dimension from lowest to highest index for each dimension, e.g. [0][0], [0][1], [1][0], [1][1]. The first port which matches all the criteria will cause the operation to be successful even if other ports in the array would also meet the criteria.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

1. Using the **check** operation in a wrong manner, e.g. check for an exception at a message-based port shall cause a test case error.
2. *AddressRef* for retrieving the sending entity shall be of type **address**, **component** or of the type provided in the address declaration of the port type of the port instance referenced in the **check** operation. No *AddressRef* shall contain the special value **null** at the time of the operation.
3. The *PortArrayRef* shall be a reference to a completely initialized port array.
4. The index redirection shall only be used when the operation is used on an any from port array construct.
5. If the index redirection is used for single-dimensional arrays, the type of the integer variable shall allow storing the highest index of the respective port array.
6. If the index redirection is used for multi-dimensional arrays, the size of the integer array or record of integer type shall exactly be the same as the dimension of the respective port array, and the its type shall allow storing the highest index (from all dimensions) of the port array.
7. If a variable referenced in the **sender** or **@index** clause is a lazy or fuzzy variable, the expression assigned to this variable is equal to the result produced by the **check** operation, i.e. later evaluation of the lazy or fuzzy variable does not lead to repeated invocation of the **check** operation.
8. If the **check** operation contains both **from** and **sender** clause, the variable or parameter referenced in the **sender** clause shall be type compatible with the template in the **from** clause.

NOTE 2: In most cases the correct usage of the check operation can be checked statically, i.e. before/during compilation.

NOTE 3: An error due to a type mismatch may happen if the types in the receive part are not present or not compatible to the types in the assignment part.

***Examples***

EXAMPLE 1: Basic check

 MyPort1.**check**(**receive**(5)); // Checks for an integer message of value 5.

 MyPort1.**check**(**receive**(charstring:?) -> **value** MyCharVar);

 // Checks for a charstring message and stores the message if the message type is charstring

 MyPort2.**check**(**getcall**(MyProc:{5, MyVar}) **from** MyPartner);

 // Checks for a call of MyProc at port MyPort2 from MyPartner

 MyPort2.**check**(**getreply**(MyProc:{5, MyVar} **value** 20));

 // Checks for a reply from procedure MyProc at MyPort2 where the returned value is 20 and

 // the values of the two out or inout parameters are 5 and the value of MyVar.

 MyPort2.**check**(**catch**(MyProc, MyTemplate(5, MyVar)));

 MyPort2.**check**(**getreply**(MyProc1:{?, MyVar} **value** \*) -> **value** MyReturnValue **param**(MyPar1,-));

 MyPort.**check**(**getcall**(MyProc:{5, MyVar}) **from** MyPartner -> **param** (MyPar1Var, MyPar2Var));

 MyPort.**check**(**getcall**(MyProc:{5, MyVar}) -> **sender** MySenderVar);

EXAMPLE 2: Check any operation

 MyPort.**check**;

 MyPort.**check**(**from** MyPartner);

 MyPort.**check**(-> **sender** MySenderVar);

EXAMPLE 3: Check on any port

 **any port**.**check;**

EXAMPLE 4: Check on any port from port array

 **type** **port** MyPort **procedure** { **inout** MyProc }

 **type** **component** MyComponent {

 **port** MyPort p[10][10];

 }

 **var** **integer** i[2];

 **any** **from** p.**check**(**catch**(MyProc, MyType:?)) -> @**index** **value** i;

 // Checking for an incoming exception of the type MyType on any port of the port array p and

 // storing the index of the port on which the matching was successful first

## 22.5 Controlling communication ports

TTCN‑3 operations for controlling message-based and procedure-based ports are presented in table 24.

Table 24: Overview of TTCN‑3 port operations

|  |
| --- |
| Port operations |
| Statement | Associated keyword or symbol |
| Clear port | **clear**  |
| Start port | **start** |
| Stop port | **stop** |
| Halt port | **halt** |
| Check the state of a port | **checkstate** |

### 22.5.1 The Clear port operation

The **clear** port operation empties incoming port queues.

***Syntactical Structure***

( *Port* | ( **all** **port** ) ) "." **clear**

***Semantic Description***

The **clear** operation removes the contents of the *incoming* queue of the specified port or of all ports of the test component performing the **clear** operation.

If a port queue is already empty then this operation shall have no action on that port.

***Restrictions***

No specific restrictions in addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15.

***Examples***

 MyPort.**clear**; // clears port MyPort

### 22.5.2 The Start port operation

The **start** operation enables sending and receiving operations on the port(s).

***Syntactical Structure***

( *Port* | ( **all** **port** ) ) "." **start**

***Semantic Description***

If a port is defined as allowing receiving operations such as **receive**, **getcall**, etc., the **start** operation clears the incoming queue of the named port and starts listening for traffic over the port. If the port is defined to allow sending operations then the operations such as **send**, **call**, **raise**, etc., are also allowed to be performed at that port.

By default, all ports of a component shall be started implicitly when a component is created. The start port operation will cause unstopped ports to be restarted by removing all messages waiting in the incoming queue.

***Restrictions***

No specific restrictions in addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15.

***Examples***

 MyPort.**start**; // starts MyPort

### 22.5.3 The Stop port operation

The **stop** operation disables sending and disallow receiving operations to match at the port(s).

***Syntactical Structure***

( *Port* | ( **all** **port** ) ) "." **stop**

***Semantic Description***

If a port is defined as allowing receiving operations such as **receive** and **getcall,** the **stop** operation causes listening at the named port to cease. If the port is defined to allow sending operations then **stop** port disallows the operations such as **send**, **call**, **raise**, etc., to be performed.

To cease listening at the port means that all receiving operations defined before the stop operation shall be completely performed before the working of the port is suspended.

***Restrictions***

No specific restrictions in addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15.

***Examples***

 MyPort.**receive** (MyTemplate1) -> **value** RecPDU;

 // the received value is decoded, matched against
 // MyTemplate1 and the matching value is stored
 // in the variable RecPDU
 MyPort.**stop**; // No receiving operation defined following the stop
 // operation is executed (unless the port is restarted
 // by a subsequent start operation)
 MyPort.**receive** (MyTemplate2); // This operation does not match and will block (assuming

 // that no default is activated)

### 22.5.4 The Halt port operation

The **halt** operation is comparable to the **stop** operation, but allows entries being already in the queue to be processed with receiving operations.

***Syntactical Structure***

( *Port* | ( **all** **port** ) ) "." **halt**

***Semantic Description***

If a port allows receiving operations such as **receive**, **trigger** and **getcall,** the **halt** operation disallows receiving operations to succeed for messages and procedure call elements that enter the port queue after performing the **halt** operation at that port. Messages and procedure call elements that were already in the queue before the **halt** operation can still be processed with receiving operations. If the port allows sending operations then **halt** port immediately disallows sending operations such as **send**, **call**, **raise**, etc. to be performed. Subsequent halt operations have no effect on the state of the port or its queue.

NOTE 1: The port **halt** operation virtually puts a marker after the last entry in the queue received when the operation is performed. Entries ahead of the marker can be processed normally. After all entries in the queue ahead of the marker have been processed, the state of the port is equivalent to the stopped state.

NOTE 2: If a port **stop** operation is performed on a halted port before all entries in the queue ahead of the marker have been processed, further receive operations are disallowed immediately (i.e. the marker is virtually moved to the top of the queue).

NOTE 3: A port **start** operation on a halted port clears all entries in the queue irrespectively if they arrived before or after performing the port **halt** operation. It also removes the marker.

NOTE 4: A port **clear** operation on a halted port clears all entries in the queue irrespectively if they arrived before or after performing the port **halt** operation. It also virtually puts the marker at the top of the queue.

***Restrictions***

No specific restrictions in addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15.

***Examples***

 MyPort.halt; // No sending allowed on Myport from this moment on;
 // processing of messages in the queue still possible.
 MyPort.receive (MyTemplate1); // If a message was already in the queue before the halt
 // operation and it matches MyTemplate1, it is processed;
 // otherwise the receive operation blocks.

### 22.5.5 The Checkstate port operation

The **checkstate** port operation allows to check the state of a port.

***Syntactical Structure***

( *Port* | ( **all** **port** ) | ( **any** **port** )) "." **checkstate** "(" *SingleExpression*")"

***Semantic Description***

The **checkstate** port operation allows to examine the state of a port. If a port is in the state specified by the parameter, the **checkstate** operation returns the Boolean value **true**. If the port is not in the specified state, the **checkstate** operation returns the Boolean value **false**. Calling the **checkstate** operation with an invalid argument leads to an error.

The checkstate operation allows to check for different dimensions of a port state. It allows to check if a port is Started, Halted or Stopped, but also if a port is Connected, Mapped or Linked (i.e. Connected or Mapped).

NOTE 1: The states Started, Halted and Stopped refer to the port states defined in the clauses F.3.1 and F.3.2. The states Connected, Mapped and Linked are related to the application of the connection operations **connect**, **disconnect**, **map** and **unmap** as defined in clause 21.1.

The **checkstate** port operation can be used with **all port** and **any port**. Using the **checkstate** operation with **any port** allows to test if at least one port of a test component is in the specified state. Using the **checkstate** operation with **all port** allows to check if all ports of a component are in the specified state.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

a) The parameter of the **checkstate** operation shall be of type **charstring** and shall have one of the following values:

a) "Started"

b) "Halted"

c) "Stopped"

d) "Connected"

e) "Mapped"

f) "Linked"

NOTE 2: Clause E.2.2.4 includes the type definition objState and the constant definitions STARTED, HALTED, STOPPED, CONNECTED, MAPPED, and LINKED. It is recommended to use the **checkstate** operation in combination with this type and these constants to ease the checking of correct usage and to improve the readability of test specs.

b) Calling the **checkstate** operation with a **charstring** parameter not listed in a) shall lead to an error.

***Examples***

 **type** **component** MyMTCType // Component type definition for an MTC

 {

 **port** MyPortType PCO1, PCO2

 }

 **type component** MyTestSystemInterface // Component type definition for a test system interface

 {

 **port** MyPortType PCO3, PCO4, PCO5;

 }

 // Test case definition

 **testcase** MyTestcase1 () **runs** **on** MyMTCType **system** MyTestSystemInterface {

 **var boolean** myPortState;

 myPortState := all port.**checkstate**("Started"); // checkstate returns true, because all

 // ports of a component are started after

 // component creation and start

 myPortState := any port.**checkstate**("Linked"); // checkstate returns false, no port is

 // either connected nor mapped

 **map**(**mtc**:PCO1, **system**:PCO3);

 myPortState := PCO1.**checkstate**("Linked"); // checkstate returns true, PCO1 is mapped

 myPortState := PCO1.**checkstate**("Mapped"); // checkstate returns true, PCO1 is mapped

 myPortState := PCO1.**checkstate**("Connected"); // checkstate returns false, PCO1 is mapped

 // and not connected

 myPortState := **any port**.**checkstate**("Mapped"); // checkstate returns true, PCO1 is mapped

 **all port**.**stop**;

 myPortState := **all port**.**checkstate**("Started"); // checkstate returns false, all ports

 // are stopped

 myPortState := PCO1.**checkstate**("Stopped"); // checkstate returns true, PCO1 is stopped

 // further testcase behaviour

 // …

 }

## 22.6 Use of any and all with ports

The keywords **any** and **all** may be used with configuration and communication operations as indicated in table 25.

Table 25: Any and All with ports

|  |  |  |
| --- | --- | --- |
| Operation | Allowed | Example |
|  | any | all |  |
| **receive, trigger, getcall, getreply, catch, check**) | yes |  | **any port.receive** |
| **connect / map** |  |  |  |
| **disconnect / unmap** |  | yes | **unmap**(**self** : **all** **port**) |
| **start, stop, clear, halt** |  | yes | **all port.start** |
| **checkstate** | yes | yes | **any port.checkstate("Started")****all port.checkstate("Connected")** |

NOTE: Ports are owned by test components and instantiated when a component is created. The keywords **any port** and **all port** address all ports owned by a test component and not only the ports known in the scope of the function or altstep that is executed on the component.