

API Technical Reference for TITAN TTCN-3 Test Executor

Jenő Balaskó

Version 11.2.1, 2025-08-05

Table of Contents

1. Introduction . . . 2
   1.1. Overview . . . 2
   1.2. Target Groups . . . 2
   1.3. Typographical Conventions . . . 2
2. Test Ports . . . 3
   2.1. Generating the Skeleton . . . 3
   2.2. Message-based Example . . . 4
   2.3. Procedure-based Example . . . 7
   2.4. Test Port Functions . . . 11
      2.4.1. Constructor and Destructor . . . 12
      2.4.2. Parameter Setting Function . . . 13
      2.4.3. Map and Unmap Functions . . . 14
      2.4.4. Start and Stop Functions . . . 17
      2.4.5. Outgoing Operations . . . 18
      2.4.6. Incoming Operations . . . 18
      2.4.7. Additional Functions and Attributes . . . 22
   2.5. Support of `address` Type . . . 23
   2.6. Provider Port Types . . . 25
   2.7. Tips and Tricks . . . 29
      2.7.1. Logging . . . 29
      2.7.2. Error Handling . . . 29
   2.8. Setting timestamps . . . 30
      2.8.1. Incoming operations . . . 31
      2.8.2. Outgoing operations . . . 31
3. External Classes . . . 33
   3.1. Example . . . 33
4. Logger Plug-ins . . . 38
   4.1. Implementing Logger Plug-ins . . . 38
   4.2. Building Logger Plug-ins . . . 39
   4.3. Event Handling . . . 40
   4.4. Execution . . . 40
5. Encoding and Decoding . . . 41
   5.1. The Common API . . . 41
      5.1.1. TTCN_EncDec . . . 41
      5.1.2. TTCN_Buffer . . . 43
      5.1.3. Invoking the Coding Functions . . . 45
   5.2. BER . . . 46
      5.2.1. Error Situations . . . 46

5.2.2. API . . . 47
5.2.3. Example . . . 48
5.3. RAW . . . 48
5.3.1. Error Situations . . . 49
5.3.2. API . . . 49
5.3.3. Example . . . 50
5.4. TEXT . . . 51
5.4.1. Error Situations . . . 52
5.4.2. API . . . 52
5.4.3. Example . . . 52
5.5. XML Encoding (XER) . . . 53
5.5.1. Error Situations . . . 54
5.5.2. API . . . 54
5.5.3. Example . . . 55
5.6. JSON . . . 55
5.6.1. Error Situations . . . 57
5.6.2. API . . . 57
5.6.3. Example . . . 57
5.7. OER . . . 58
5.7.1. Error Situations . . . 58
5.7.2. API . . . 59
5.7.3. Example . . . 59
5.8. PER . . . 60
5.8.1. Error Situations . . . 60
5.8.2. API . . . 60
5.8.3. Example . . . 61
6. Mapping TTCN–3 Data Types to C++ Constructs . . . 63
6.1. Mapping of Names and Identifiers . . . 63
6.2. Namespaces . . . 64
6.3. Predefined TTCN–3 Data Types . . . 64
6.3.1. `Integer` . . . 64
6.3.2. `Float` . . . 67
6.3.3. `Boolean` . . . 69
6.3.4. `Verdicttype` . . . 70
6.3.5. `Bitstring` . . . 72
6.3.6. `Hexstring` . . . 76
6.3.7. `Octetstring` . . . 80
6.3.8. `Char` . . . 84
6.3.9. `Charstring` . . . 84
6.3.10. `Universal char` . . . 89
6.3.11. `Universal charstring` . . . 89

6.3.12. Object Identifier Type . . . 98
6.3.13. Component References . . . 100
6.3.14. Empty Types . . . 101
6.4. Compound Data Types . . . 103
6.4.1. Record and Set Type Constructs . . . 103
6.4.2. Union Type Construct . . . 106
6.4.3. Record of Type Construct . . . 109
6.4.4. `Set of` Type Construct . . . 113
6.4.5. Enumerated Types . . . 113
6.4.6. The `address` Type . . . 116
6.4.7. The TTCN-3 open type . . . 116
6.5. Predefined Functions . . . 117
6.5.1. `Integer` to character . . . 117
6.5.2. Character to `integer` . . . 117
6.5.3. `Integer` to universal character . . . 118
6.5.4. Universal character to `integer` . . . 118
6.5.5. `Bitstring` to `integer` . . . 118
6.5.6. `Hexstring` to `integer` . . . 118
6.5.7. `Octetstring` to `integer` . . . 118
6.5.8. `Charstring` to `integer` . . . 118
6.5.9. `Integer` to `bitstring` . . . 118
6.5.10. `Integer` to `hexstring` . . . 118
6.5.11. `Integer` to `octetstring` . . . 118
6.5.12. `Integer` to `charstring` . . . 119
6.5.13. Length of string Type . . . 119
6.5.14. Number of elements in a structured type . . . 119
6.5.15. The `IsPresent` Function . . . 119
6.5.16. The `IsChosen` Function . . . 119
6.5.17. The `regexp` Function . . . 119
6.5.18. `Bitstring` to `charstring` . . . 120
6.5.19. `Hexstring` to `charstring` . . . 120
6.5.20. `Octetstring` to character string . . . 120
6.5.21. Character string to `octetstring` . . . 120
6.5.22. `Bitstring` to `hexstring` . . . 120
6.5.23. `Hexstring` to `octetstring` . . . 120
6.5.24. `Bitstring` to `octetstring` . . . 120
6.5.25. `Hexstring` to `bitstring` . . . 120
6.5.26. `Octetstring` to `hexstring` . . . 120
6.5.27. `Octetstring` to `bitstring` . . . 121
6.5.28. `Integer` to `float` . . . 121
6.5.29. `Float` to `integer` . . . 121

6.5.30. The Random Number Generator Function . . . 121
6.5.31. The Substring Function . . . 121
6.5.32. Character string to `float` . . . 121
6.5.33. The Replace Function . . . 122
6.5.34. Octetstring to character string . . . 122
6.5.35. Character string to octetstring . . . 122
6.5.36. The `Decompose` Function . . . 122
6.5.37. Additional Non-Standard Functions . . . 122
6.6. Using the Signature Classes . . . 123
6.6.1. The Representation of the Input Parameters . . . 124
6.6.2. The Output Parameters and Return Value . . . 124
6.6.3. Representation of Signature Exceptions . . . 125
6.7. Object references . . . 126
7. Tips & Troubleshooting . . . 128
7.1. Migrating Existing C++ Code to the Naming Rules of Version 1.7 . . . 128
7.2. Using External C++ Functions in TTCN–3 Test Suites . . . 128
7.2.1. Example TTCN–3 Module (MyExample.ttcn) . . . 129
7.3. Logging in Test Ports or External Functions . . . 130
7.3.1. Unbuffered Mode . . . 130
7.3.2. Buffered Mode . . . 131
7.3.3. Logging Format of TTCN-3 Values and Templates . . . 133
7.3.4. Logging sensitive data . . . 134
7.3.5. Examples . . . 134
7.4. Error Recovery during Test Execution . . . 135
7.5. Using UNIX Signals . . . 136
7.6. Mixing C and C++ Modules . . . 136
8. References . . . 137
9. Abbreviations . . . 138

Abstract

This document describes detailed information on the TITAN Application Programming Interface (API) on C++ level, advanced TTCN–3 programming, and background information on the TITAN TTCN–3 Test Executor project.

Copyright

Copyright (c) 2000-2026 Ericsson Telecom AB
All rights reserved. This program and the accompanying materials are made available under the terms of the Eclipse Public License v2.0 that accompanies this distribution, and is available at

https://www.eclipse.org/org/documents/epl-2.0/EPL-2.0.html.

Disclaimer

The contents of this document are subject to revision without notice due to continued progress in methodology, design and manufacturing. Ericsson shall have no liability for any error or damage of any kind resulting from the use of this document.

Chapter 1. Introduction

1.1. Overview

This document describes the TITAN API on C++ level. It is intended for users who write test port implementation, external function implementation in language C++ and want to use the available resources of TITAN.

Detailed information can be found on the following topics:

- test ports, the communication link between the TITAN Executor and System Under Test (SUT);
- built-in encoding and decoding functions;
- TTCN-3 data mapping to C++ constructs;
- troubleshooting for common TTCN-3 related issues and problems.

1.2. Target Groups

This document is intended for advanced users of the TITAN API on C++ level.

1.3. Typographical Conventions

This document uses the following typographical conventions:

Bold is used to represent graphical user interface (GUI) components such as buttons, menus, menu items, dialog box options, fields and keywords, as well as menu commands. Bold is also used with '+' to represent key combinations. For example, **Ctrl+Click**

The '/' character is used to denote a menu and sub-menu sequence. For example, **File / Open**.

`Monospaced` font is used represent system elements such as command and parameter names, program names, path names, URLs, directory names and code examples.

`Bold monospaced` font is used for commands that must be entered at the Command Line Interface (CLI), For example, `ttcn3_start`

Chapter 2. Test Ports

The C++ source code generated by the Compiler is protocol independent, that is, it does not contain any device specific operations. To provide the connection between the executable test suite and SUT, that is, the physical interface of the test equipment[1], a so-called Test Port is needed.

The Test Port is a software library written in C++ language, which is linked to the executable test program. It maps the device specific operations to function calls specified in an API. This chapter describes the Test Port API in details.

2.1. Generating the Skeleton

The functions of Test Ports must be written by the user who knows the interface between the executable test suite and the test equipment. In order to make this development easier, the Compiler can generate Test Port skeletons. A Test Port belongs to one certain TTCN–3 port type, so the skeleton is generated based on port type definitions.

A Test Port consists of two parts. One part is generated automatically by the Compiler, and it is put into the generated C++ code. The user has nothing to do with this part.

The other part is a C++ class, which is written mainly by the user. This class can be found in a separate C++ header and source file (their suffixes are `.hh` and `.cc`, respectively). The names of the source files and the C++ class are identical to the name of the port type. Please note that the name mapping rules described in Mapping of Names and Identifiers also apply to these class and file names.

During the translation, when the Compiler encounters a port type definition and the `t` command line switch is used, it checks whether the header and source files of Test Port exist in its working directory. If none of them can be found there, the compiler generates the skeleton header and source files for the corresponding test port automatically. This means, once you have generated (and possibly modified) a skeleton, it will never be overwritten. If you want to re-generate the skeleton, you must rename or remove the existing one.

If the list of message types/signatures of a TTCN-3 port type changes, the list of the Test Port class member functions also needs to change. If the Test Port skeleton has been generated, it will not be modified, resulting in build errors (C++ compile errors like "cannot declare variable of abstract type"/"virtual functions are pure", or linker errors). In this case, the Test Port skeleton files should be renamed/moved, the skeleton generated, and any user-written code should be copied back into the newly generated source files.

If you have defined a TTCN–3 port type that you intend to use for internal communication only (that is, for sending and receiving messages between TTCN–3 test components), you do not need to generate and compile an empty Test Port skeleton for that port type. Adding the attribute with `{extension "internal"}` to the port type definition in the TTCN–3 module disables the generation and use of a Test Port for the port type.

WARNING	In this case you must not link the object file obtained from a previous Test Port skeleton to your executable test suite.

In the following we introduce two port type definitions: one for a message based and another one for a procedure based port. In our further examples we will refer to the test port skeletons generated according to these definitions given within the module called `MyExample`.

2.2. Message-based Example

The definition of `MyMessagePort`:

```
type port MyMessagePort message
{
  in octetstring;
  out integer;
  inout charstring;
};
```

That is, the types integer and charstring can be sent, and octetstring and charstring can be received on port `MyMessagePort`.

The generated skeleton header file (that is, `MyMessagePort.hh`) will look as follows:

```
// This Test Port skeleton header file was generated by the
// TTCN-3 Compiler of the TTCN-3 Test Executor version 7/CAX 105 7730 R2A
// for U-ERICSSON\ethbaat (ethbaat@HU-00000670) on Wed Nov 18 14:49:55 2020

// Copyright (c) 2000-2026 Ericsson Telecom AB

// You may modify this file. Add your attributes and prototypes of your
// member functions here.

#ifndef MyMessagePort_HH
#define MyMessagePort_HH

#include "MyExample.hh"

namespace MyExample {

class MyMessagePort : public MyMessagePort_BASE {
public:
        MyMessagePort(const char *par_port_name = NULL);
        ~MyMessagePort();

        void set_parameter(const char *parameter_name,
                const char *parameter_value);

private:
        /* void Handle_Fd_Event(int fd, boolean is_readable,
                boolean is_writable, boolean is_error); */
        void Handle_Fd_Event_Error(int fd);
        void Handle_Fd_Event_Writable(int fd);
        void Handle_Fd_Event_Readable(int fd);
        /* void Handle_Timeout(double time_since_last_call); */
protected:
        void user_map(const char *system_port, Map_Params& params);
        void user_unmap(const char *system_port, Map_Params& params);

        void user_start();
        void user_stop();

        void outgoing_send(const INTEGER& send_par);
        void outgoing_send(const CHARSTRING& send_par);
};

} /* end of namespace */

#endif
```

And the generated skeleton source file, that is, `MyMessagePort.cc`, will be the following:

```
// This Test Port skeleton source file was generated by the
```

```
// TTCN-3 Compiler of the TTCN-3 Test Executor version  7/CAX 105 7730 R2A
// for U-ERICSSON\ethbaat (ethbaat@HU-00000670) on Wed Nov 18 14:49:55 2020

// Copyright (c) 2000-2026 Ericsson Telecom AB

// You may modify this file. Complete the body of empty functions and
// add your member functions here.

#include "MyMessagePort.hh"

namespace MyExample {

MyMessagePort::MyMessagePort(const char *par_port_name)
        : MyMessagePort_BASE(par_port_name)
{

}

MyMessagePort::~MyMessagePort()
{

}

void MyMessagePort::set_parameter(const char * /*parameter_name*/,
        const char * /*parameter_value*/)
{

}

/*void MyMessagePort::Handle_Fd_Event(int fd, boolean is_readable,
        boolean is_writable, boolean is_error) {}*/

void MyMessagePort::Handle_Fd_Event_Error(int /*fd*/)
{

}

void MyMessagePort::Handle_Fd_Event_Writable(int /*fd*/)
{

}

void MyMessagePort::Handle_Fd_Event_Readable(int /*fd*/)
{

}

/*void MyMessagePort::Handle_Timeout(double time_since_last_call) {}*/

void MyMessagePort::user_map(const char * /*system_port*/, Map_Params& /*params*/)
{
```

```
}

void MyMessagePort::user_unmap(const char * /*system_port*/, Map_Params& /*params*/)
{

}

void MyMessagePort::user_start()
{

}

void MyMessagePort::user_stop()
{

}

void MyMessagePort::outgoing_send(const INTEGER& /*send_par*/)
{

}

void MyMessagePort::outgoing_send(const CHARSTRING& /*send_par*/)
{

}

} /* end of namespace */
```

2.3. Procedure-based Example

The definition of `MyProcedurePort` in module `MyExample`:

```
type port MyProcedurePort procedure
{
  in inProc;
  out outProc;
  inout inoutProc;
};
```

The signature definitions are imported from a module called `MyExample`, `noblock` is not used and exceptions are used so that every member function of the port class is generated for this example. If the keyword `noblock` is used the compiler will optimize code generation by not generating outgoing reply, incoming reply member functions and their argument types. If the signature has no exception outgoing raise, incoming exception member functions and related types will not be generated.

The port type `MyProcedurePort` can handle `call`, `getreply` and `catch` operations referencing the signatures `outProc` and `inoutProc`, and it can handle `getcall`, `reply` and `raise` operations referencing the signatures `inProc` and `inoutProc`.

The generated skeleton header file (that is, `MyProcedurePort.hh`) will look as follows:

```
// This Test Port skeleton header file was generated by the
// TTCN-3 Compiler of the TTCN-3 Test Executor version 1.7.pre4 build 4
// for Csaba Feher (ecsafeh@ehubuux110) on Tue Jul 29 18:53:35 2008

// Copyright (c) 2000-2026 Ericsson Telecom AB

// You may modify this file. Add your attributes and prototypes of your
// member functions here.

#ifndef MyProcedurePort_HH
#define MyProcedurePort_HH

#include "MyExample.hh"

namespace MyExample {

class MyProcedurePort : public MyProcedurePort_BASE {
public:
    MyProcedurePort(const char *par_port_name = NULL);
    ~MyProcedurePort();

    void set_parameter(const char *parameter_name,
        const char *parameter_value);

private:
    /* void Handle_Fd_Event(int fd, boolean is_readable,
        boolean is_writable, boolean is_error); */
    void Handle_Fd_Event_Error(int fd);
    void Handle_Fd_Event_Writable(int fd);
    void Handle_Fd_Event_Readable(int fd);
    /* void Handle_Timeout(double time_since_last_call); */
protected:
    void user_map(const char *system_port, Map_Params& params);
    void user_unmap(const char *system_port, Map_Params& params);

    void user_start();
    void user_stop();

    void outgoing_call(const outProc_call& call_par);
    void outgoing_call(const inoutProc_call& call_par);
    void outgoing_reply(const inProc_reply& reply_par);
    void outgoing_reply(const inoutProc_reply& reply_par);
};

} /* end of namespace */

#endif
```

The generated skeleton source file for `MyProcedurePort` (that is, `MyProcedurePort.cc`) will be the following:

```
// This Test Port skeleton source file was generated by the
// TTCN-3 Compiler of the TTCN-3 Test Executor version 1.7.pre4 build 4
// for Csaba Feher (ecsafeh@ehubuux110) on Tue Jul 29 18:53:35 2008

// Copyright (c) 2000-2026 Ericsson Telecom AB

// You may modify this file. Complete the body of empty functions and
// add your member functions here.

#include "MyProcedurePort.hh"

namespace MyExample {

MyProcedurePort::MyProcedurePort(const char *par_port_name)
    : MyProcedurePort_BASE(par_port_name)
{

}

MyProcedurePort::~MyProcedurePort()
{

}

void MyProcedurePort::set_parameter(const char *parameter_name,
    const char *parameter_value)
{

}

/*void MyProcedurePort::Handle_Fd_Event(int fd, boolean is_readable,
    boolean is_writable, boolean is_error) {}*/

void MyProcedurePort::Handle_Fd_Event_Error(int fd)
{

}

void MyProcedurePort::Handle_Fd_Event_Writable(int fd)
{

}

void MyProcedurePort::Handle_Fd_Event_Readable(int fd)
{

}

/*void MyProcedurePort::Handle_Timeout(double time_since_last_call) {}*/
```

```
void MyProcedurePort::user_map(const char *system_port, Map_Params& params)
{

}

void MyProcedurePort::user_unmap(const char *system_port, Map_Params& params)
{

}

void MyProcedurePort::user_start()
{

}

void MyProcedurePort::user_stop()
{

}

void MyProcedurePort::outgoing_call(const outProc_call& call_par)
{

}

void MyProcedurePort::outgoing_call(const inoutProc_call& call_par)
{

}

void MyProcedurePort::outgoing_reply(const inProc_reply& reply_par)
{

}

void MyProcedurePort::outgoing_reply(const inoutProc_reply& reply_par)
{

}

} /* end of namespace */
```

2.4. Test Port Functions

This section summarizes all possible member functions of the Test Port class. All of these functions exist in the skeleton, but their bodies are empty.

The identical functions of both port types are:

- the constructor and the destructor

- the parameter setting function
- the map and unmap function
- the start and stop function
- descriptor event and timeout handler(s)
- some additional functions and attributes

The functions above will be described using an example of message based ports (`MyMessagePort`, also introducing the functions specific to message based port types). Using these functions is identical (or very similar) in procedure based Test Ports.

Functions specific to message based ports:

- send functions: outgoing send
- incoming functions: incoming message
- Functions specific to procedure based ports:
- outgoing functions: outgoing call, outgoing reply, outgoing raise
- incoming functions: incoming call, incoming reply, incoming exception

Both test port types can use the same logging and error handling mechanism, and the handling of incoming operations on port `MyProcedurePort` is similar to receiving messages on port `MyMessagePort` (regarding the event handler).

2.4.1. Constructor and Destructor

The Test Port class belongs to a TTCN–3 port type, and its instances implement the functions of the port instances. That is, each Test Port instance belongs to the port of a TTCN–3 test component. The number of TTCN–3 component types, port types and port instances is not limited; you may have several Test Port classes and several instances of a given Test Port class in one test suite.

The Test Port instances are global and static objects. This means, their constructor and destructor is called before and after the test execution (that is, before the main function starts and after the main function finishes). The name of a Test Port object is composed of the name of the corresponding component type and the name of the port instance within the component type.

In case of parallel test execution, each TTCN–3 test component process has its own Test Port instances of all ports defined in all component types within the entire test suite. Of course, only the Test Ports of the active component type are used, the member functions of other inactive Test Port instances (except constructor and destructor) will never be called. Since all Test Port instances are static, their constructor and destructor is called only once on each host and in the Host Controller process (outside its main function). The test component processes (that is, the child processes of Host Controller) will get a copy of the initialized Test Port instances and no constructor will be called again.

The Test Port class is derived from an abstract base class which can be found in the generated code. The base class implements, for instance, the queue of incoming messages.

The constructor takes one parameter containing the name of the port instance in a NUL character

terminated string. This string shall be passed further to the constructor of the base class as it can be found in the skeleton code. The default argument for the test port name is a NULL pointer, which is used when the test port object is a member of a port array.

WARNING	In case of port arrays the name of the test port is set after the constructor is completed. So the name of the test port should not be used in the constructor. The port name is always set correctly when any other member function is called.

The destructor does nothing by default. If some dynamically allocated attributes are added to the test port class, one should free the memory and release all resources in the destructor.

WARNING	As the constructor and the destructor are called outside of main function, be careful when writing them. For instance, there is no way for error recovery; `exit(3)` call may result in a segmentation fault. If file descriptors are opened (and kept opened) here, the `fork(2)` system call of Host Controller will only multiply the file descriptors and not the kernel file structure. Therefore system and library calls should be avoided here.

2.4.2. Parameter Setting Function

Test Port parameters[2] shall contain information which is independent from the TTCN3 test suite. These values shall not be used in the test suite at all. You can define them as TTCN–3 constants or module parameters, but these definitions are useless and redundant, and they must always be present when the Test Port is used.

For instance, using Test Port parameters can be used to convey configuration data (that is, some options or extra information that is necessary for correct operation) or lower protocol layer addresses (for example, IP addresses).

Test Port parameters shall be specified by the user of executable tests in section `[TESTPORT_PARAMETERS]` of the run-time configuration file (see section `[TESTPORT_PARAMETERS]` in Programmer's Technical Reference). The parameters are maintained for each test port instance separately; wildcards can be used as well. In the latter case the parameter is passed to all Test Port matching the wildcard.

Each Test Port parameter must have a name, which must be unique within the Test Port only. The name must be a valid identifier, that is, it must begin with a letter and must contain alphanumerical characters only.

All Test Port parameter values are interpreted by the test executor as character strings. Quotation marks must be used when specifying the parameter values in the configuration file. The interpretation of parameter values is up to you: you can use some of them as symbolic values, numbers, IP addresses or anything that you want.

Before the test execution begins, all parameters belonging to the Test Port are passed to the Test Port by the runtime environment of the test executor using the function `set_parameter`. It is a virtual function, that is, this function may be removed from the header and source file if there are no parameters. Its default ancestor does nothing and ignores all parameters.

Each parameter is passed to the Test Port one-by-one separately[3], the two arguments of `set_parameter` contain the name and value of the corresponding parameter, respectively, in NUL character terminated strings. If the parameter values are needed in further operations, backup copies must be made of them because the string will disappear after the calling function returns.

It is warmly recommended that the Test Port parameter handling functions be fool-proof. For instance, the Test Port should produce a proper error message (for example by calling `TTCN_error`) if a mandatory parameter is missing instead of causing segmentation fault. Repeated setting of the same parameter should produce warnings for the user (for example by using the function `TTCN_warning`) and not memory leaks.

NOTE

On the MTC, in both single and parallel modes, the handling of Test Port parameters is a bit different from that on PTCs. The parameters are passed only to active ports, but the component type of MTC (thus the set of active ports) depends on the `runs on` clause of the test case that is currently being executed. It would be difficult for the runtime environment to check at the beginning of each test case whether the corresponding MTC component type has already been active during a previous test case run. Therefore all Test Port parameters belonging to the active ports of the MTC are passed to the `set_parameter` function at the beginning of every test case. The Test Ports of MTC shall be prepared to receive the same parameters several times (with the same values, of course) if more than one test case is being executed.

If system related Test Port parameters are used in the run-time configuration file (that is, the keyword `system` is used as component identifier), the parameters are passed to your Test Port during the execution of TTCN–3 `map` operations, but before calling your `user_map` function. Please note that in this case the port identifier of the configuration file refers to the port of the test system interface that your port is mapped to and not the name of your TTCN–3 port.

The name and exact meaning of all supported parameters must be specified in the user documentation of the Test Port.

2.4.3. Map and Unmap Functions

The run-time environment of the TTCN–3 executor knows nothing about the communication towards SUT, thus, it is the user's responsibility to establish and terminate the connection with SUT. The TTCN–3 language uses two operations to control these connections, `map` and `unmap`.

For this purpose, the Test Port class provides two member functions, `user_map` and `user_unmap`. These functions are called by the test executor environment when performing TTCN–3 `map` and `unmap` operations, respectively.

The `map` and `unmap` operations take two pairs of component references and ports as arguments. These operations are correct only if one of the arguments refer to a port of a TTCN–3 test component while the other port corresponds to SUT. This aspect of correctness is verified by the run-time environment, but the existence of a system port is not checked.

The port names of the system are converted to `NUL` character terminated strings and passed to functions `user_map` and `user_unmap` as parameters. Unlike other identifiers, the underscore characters in these port names are not translated.

If these system port names should be reused later, the entire strings (and not only the pointers) must be saved in the internal memory structures since the string values will disappear after the `user_map` or `user_unmap` finishes.

NOTE	in TTCN–3 it is not allowed to map a test component port to several system ports at the same time. The run-time environment, however, is not so strict and allows this to handle transient states during configuration changes. In this case messages can not be sent to SUT even with explicit addressing, but the reception of messages is permitted. When putting messages into the input queue of the port, it is not important for the test executor (even for the TTCN–3 language) which port of the system the message is received from.

The execution of TTCN–3 test component that requested the mapping or unmapping is suspended until your `user_map` or `user_unmap` functions finish. Therefore it is not allowed to block unnecessarily the test execution within these functions.

When the Test Port detects an error situation during the establishment or termination of the physical connection towards the SUT, the function `TTCN_error` shall be used to indicate the failure. If the error occurs within `user_map` the run-time environment will assume that the connection with SUT is not established thus it will not call `user_unmap` to destroy the mapping during the error recovery procedure. If `user_map` fails, it is the Test Port writer's responsibility to release all allocated resources and bring the object variables into a stable state before calling `TTCN_error`. Within `user_unmap` the errors should be handled in a more robust way. After a minor failure it is better to issue a warning and continue the connection termination instead of panicking. `TTCN_error` shall be called only to indicate critical errors. If `user_unmap` is interrupted with an error the run-time environment assumes that the mapping has been terminated, that is, `user_unmap` will not be called again.

NOTE	if either `user_map` or `user_unmap` fails, the error is indicated on the initiator test component as well; that is, the respective map or `unmap` operation will also fail and error recovery procedure will start on that component.

Parameters of the Map and Unmap Functions

Parameters can be sent to the `user_map` and `user_unmap` functions from TTCN code using the `param` clause of the `map` and `unmap` operations.

The 'user_map` and `user_unmap` functions have a parameter of type `Map_Params`, which contains the string representations of the `in` and `inout` parameters of the `map`/`unmap` operation. The string representations of `out` parameters are empty strings (as these are considered as being `unbound` at the beginning of the `map`/`unmap` operation). After the `user_map` or `user_unmap` function ends and the mapping/unmapping is concluded, the final values (string representations) of `out` and `inout` parameters in the `Map_Params` object are sent back to the mapping/unmapping requestor.

The following member functions can be used to obtain or set data in the `Map_Params` object:

```
unsigned int get_nof_params() const
```

Returns the number of parameters in the object. This will either be zero (if the `map` or `unmap` operation had no `param` clause) or the number of parameters specified in the system port type definition's `map param` or `unmap param` clause.

```
const CHARSTRING& get_param(unsigned int p_index) const
```

Returns the string representation of the parameter at index `p_index`. This method shall be used to retrieve the values of `in` and `inout` parameters. The parameter indices start at 0. The order of the parameters is the same as their order of declaration. Default values of parameters are automatically set by the runtime environment before the `user_map`/`user_unmap` call. The string representations retrieved with this function can be converted back to the parameter's TTCN-3 type with the predefined function `string_to_ttcn`.

```
void set_param(unsigned int p_index, const CHARSTRING& p_param)
```

Sets the string representation of the parameter at index `p_index` to the string `p_param`. This method shall be used to set the final values of `out` and `inout` parameters. The string representation of a TTCN-3 value can be obtained using the predefined function `ttcn_to_string`. If the final value of an `out` or `inout` parameter is an empty string, then the variable used as parameter will remain unchanged. Otherwise its new value will be calculated by applying `string_to_ttcn` on the string value set in the `user_map` or `user_unmap` function (this could cause dynamic test case errors if the string representation is invalid).

Usage example:

Port type:

```
type port MyPort message {
  ...
  map param(in MyInParType in_par, inout MyInOutParType inout_par, out MyOutParType
out_par)
}
```

`user_map` function in port implementation:

```
void MyPort::user_map(const char * system_port, Map_Params& params)
{
  if (params.get_nof_params() != 0) {
    // there were map parameters

    // extract 'in' and 'inout' parameters
    MyInParType in_par;
    string_to_ttcn(params.get_param(0), in_par);
    MyInOutParType inout_par;
    string_to_ttcn(params.get_param(1), inout_par);
    MyOutParType out_par; // remains unbound

    // do mapping
    ...

    // update 'out' and 'inout' parameters
    params.set_param(1, ttcn_to_string(inout_par));
    params.set_param(2, ttcn_to_string(out_par));
  }
  else {
    // there were no map parameters

    // do mapping
    ...
  }
}
```

2.4.4. Start and Stop Functions

The Test Port class has two member functions: `user_start` and `user_stop`. These functions are called when executing `port start` and `port stop` operations, respectively. The functions have no parameters and return types.

These functions are called through a stub in the base class, which registers the current state of the port (whether it is started or not). So `user_start` will never be called twice without calling `user_stop` or vice versa.

WARNING	From version 1.2.pl0 on (according to the latest TTCN–3 standard) all ports of test components are started implicitly immediately after creation. Such operations must not be put in a `user_start` function blocking the execution for a longer period. This not only hangs the new PTC but the also component that performed the `create` operation (usually the MTC). All ports are stopped at the end of test cases or at PTC termination, even if `stop` statements are missing.

In functions `user_start` and `user_stop` the device should be initialized or shut down towards SUT (that is, the communications socket). Also the event handler should be installed or uninstalled (see later).

2.4.5. Outgoing Operations

Outgoing operations are `send` (specific to message based ports); `call`, `reply`, and `raise` (specific to procedure based ports).

Send Functions

The Test Port class has an overloaded function called `outgoing_send` for each outgoing message type. This function will be called when a message is sent on the port and it should be routed to the system (that is, SUT) according to the addressing semantics[4] of TTCN–3. The messages (implicitly or explicitly) addressed to other test components are handled inside the test executor; the Test Ports have nothing to do with them. The function `outgoing_send` will be also called if the port has neither connections nor mappings, but a message is sent on it.

The only parameter of `outgoing_send` contains a read-only reference to the message in the internal data representation format of the test executor. The access methods for internal data types are described in XML Encoding (XER). The test port writer should encode and send the message towards SUT. For information on how to use the standard encoding functions like BER, please consult Logger Plug-ins. Sending a message on a not started port causes a dynamic test case error. In this case outgoing_send will not be called.

Call, Reply and Raise Functions

The procedure based Test Port class has overloaded functions called `outgoing_call`, `outgoing_reply` and `outgoing_raise` for each `call`, `reply` and `raise` operations, respectively. One of these functions will be called when a port-operation is addressing the system (that is, SUT using the to `system` statement).

The only parameter of these functions is an internal representation of the signature parameters (and possibly its return value) or the exceptions it may raise. The signature classes are described in Using the Signature Classes.

2.4.6. Incoming Operations

Incoming operations are `receive` incoming messages (specific to message based ports); `call`, `reply` and `exception` (specific to procedure based ports).

Descriptor Event and Timeout Handlers

The handling of incoming messages (or operations) is more difficult than sending. The executable test program has two states. In the first state, it executes the operations one by one as specified in the test suite (for example, it evaluates expressions, calls functions, sends messages, etc.). In the other state it waits for the response from SUT or for a timer to expire. This happens when the execution reaches a blocking statement, that is, one of a stand-alone `receive`, `done`, `timeout` statements or an `alt` construct.

After reaching a blocking statement, the test executor evaluates the current snapshot of its timer and port queues and tries to match it with the reached statements and templates. If the matching fails, the executor sleeps until something happens to its timers or ports. After waking up, it re-evaluates its snapshot and tries to match it again. The last two steps are repeated until the executor

finds the first matching statement. If the test executor realizes that its snapshot can never match the reached TTCN–3 statements, it causes a dynamic test case error. This mechanism prevents it from infinite blocking.

The test executor handles its timers itself, but it does not know anything about the communication with SUT. So each Test Port instance should inform the snapshot handler of the executor what kind of event the Test Port is waiting for. The event can be either the reception of data on one or more file descriptors or a timeout (when polling is used) or both of them.

When the test executor reaches a blocking statement and any condition – for which the Test Port waits – is fulfilled, the event handler will be called. First one has to get the incoming message or operation from the operating system. After that, one has to decode it (and possibly decide its type). Finally, if the internal data structure is built, one has to put it into the queue of the port. This can be done using the member function `incoming_message` if it is a message, and using `incoming_call`, `incoming_reply` or `incoming_exception` if it is an operation.

The execution must not be blocked in event handler functions; these must return immediately when the message or operation processing is ready. In other words, always use non-blocking `recv()` system calls. In the case when the messages are fragmented (for instance, when testing TCP based application layer protocols, such as HTTP), intermediate buffering should be performed in the Test Port class.

Event and timeout handling interface introduced in TITAN version 1.7.pl4

This descriptor event and timeout handling interface is the preferred interface for new Test Port development.

There are two possibilities to be notified about available events:

- Either the `Handle_Fd_Event` function has to be implemented, or
- `Handle_Fd_Event_Readable`, `Handle_Fd_Event_Writable`, and `Handle_Fd_Event_Error`.

Using `Handle_Fd_Event` allows receiving all events of a descripor in one function call. Using the other three event handler functions allows creating a more structured code.

All these functions are virtual. The unused event handler functions have to be left un-overridden. (When using the second alternative and the Test Port does not wait for all types of events (readable, writable, error) the handlers of the events – for which the Test Port does not wait – can be left un-overridden.)

The following functions can be used to add events to and remove events from the set of events for which the Test Port waits:

```
void Handler_Add_Fd(int fd, Fd_Event_Type event_mask = EVENT_ALL);
void Handler_Add_Fd_Read(int fd);
void Handler_Add_Fd_Write(int fd);
void Handler_Remove_Fd(int fd, Fd_Event_Type event_mask = EVENT_ALL);
void Handler_Remove_Fd_Read(int fd);
void Handler_Remove_Fd_Write(int fd);
```

The first parameter in all of these functions is the file descriptor. Possible values of the event_mask are EVENT_RD, EVENT_WR, EVENT_ERR and combinations of these using bitwise or: "|".

Timeout notification can be received with the Handle_Timeout function. The parameter of the function indicates the time elapsed in seconds since its last call of this function or the latest modification of the timer (whichever occurred later).

The timer can be set with the following function:

```
void Handler_Set_Timer(double call_interval, boolean is_timeout = TRUE,
    boolean call_anyway = TRUE, boolean is_periodic = TRUE);
```

call_interval is measured in seconds and specifies the time after which the Handle_Timeout function will be called. To stop the timer call_interval value: 0.0 has to be given.

is_timeout specifies if the timer has to be stopped when event handler is called. call_anyway is meaningful when is_timeout is set to TRUE. In this case call_anyway indicates if the Handle_Timeout function has to be called when event handler is called before the timer would expire. If call_anyway is TRUE the timeout handler will be called after the call of the event handlers and the timer will be stopped. is_periodic indicates if the timer has to be restarted instead of stopping when timer expires or event handler is called and is_timeout and call_anyway are both TRUE.

Event handler for Test Ports developed for 1.7pl3 and earlier versions of TITAN

There is only one event handler function in each Test Port class called Event_Handler, which is a virtual member function. The run-time environment calls it when an incoming event arrives. You can install or uninstall the event handler by calling the following inherited member functions:

```
void Install_Handler(const fd_set *read_fds, const fd_set *write_fds,
                     const fd_set *error_fds, double call_interval);
void Uninstall_Handler();
```

Install_Handler installs the event handler according to its parameters. It takes four arguments, three pointers pointing to bitmasks of file descriptors and a timeout value. Some of the parameters can be ignored, but ignoring all at the same time is not permitted.

The bitmasks are interpreted in the same way as in the select system call. They can be set using the macros FD_ZERO, FD_SET and FD_CLR. If the pointer is NULL, the bitmask is treated as zero. For further details see the manual page of select(2) or select(3).

The call interval value is measured in seconds. It means that the event handler function will be called when the time elapsed since its last call reaches the given value. This parameter is ignored when its value is set to zero or negative.

If you want to change your event mask parameters, you may simply call the function Install_Handler again (calling of Uninstall_Handler is not necessary).

Uninstall_Handler will uninstall your previously installed event handler. The stop port operation

also uninstalls the event handler automatically. The event handler may be installed or uninstalled in any Test Port member function, even in the event handler itself.

The prototype of the event handler function is the following:

```
void Event_Handler(const fd_set *r_fds, const fd_set *w_fds,
                   const fd_set *e_fds, double time_since_last_call);
```

The function `Event_Handler` has four parameters. The first three of them are pointers to bitmasks of file descriptors as described above. They are the bitwise AND combination of bitmasks you have given to `Install_Handler` and the bitmasks given back by the last call of select. They can be useful when waiting for data from many file descriptors, for example when handling more than one SUT mappings simultaneously, because there is no need to issue a select call again within the event handler.

NOTE the pointers can be never NULL, they point to a valid memory area even if there are no file descriptors set in the bitmask. The last parameter contains the time elapsed since the last call of the event handler measured in seconds. This value is always calculated even if the call interval has not been set. If the `Event_Handler` is called the first time since its last installation, the time is measured from the call of `Install_Handler`.[5]

Receiving messages

The member function `incoming_message` of message based ports can be used to put an incoming message in the queue of the port. There are different polymorphic functions for each incoming message type. These functions are inherited from the base class. The received messages are logged when they are put into the queue and not when they are processed by the test suite[6].

In our example the class `MyMessagePort_BASE` has the following member functions:

```
incoming_message(const OCTETSTRING& incoming_par);
incoming_message(const CHARSTRING& incoming_par);
```

Receiving calls, replies and exceptions

Receiving operations on procedure based ports is similar to receiving messages on message based ports. The difference is that there are different overloaded incoming functions for call, reply and raise operations called `incoming_call`, `incoming_reply` and `incoming_exception`, respectively. The event handler (when called) must recognize the type of operation on receiving and call one of these functions accordingly with one of the internal representations of the signature (see Additional Non-Standard Functions).

In the example[7] the class `MyProcedurePort_BASE` has the following member functions for incoming operations:

```
incoming_call(const MyExample::inProc_call& incoming_par);
incoming_call(const MyExample::inoutProc_call& incoming_par);
incoming_reply(const MyExample::outProc_reply& incoming_par);
incoming_reply(const MyExample::inoutProc_reply& incoming_par);
incoming_exception(const MyExample::outProc_exception& incoming_par);
incoming_exception(const MyExample::inoutProc_exception& incoming_par);
```

For example, if the event handler receives a call operation that refers to the signature called `inoutProc`, it has to fill the parameters of an instance of the class `inoutProc_call` with the received data. Then it has to call the function `incoming_call` with this object to place the operation into the queue of the port.

The following table shows the relation between the direction of the message type or signature in the port type definition and the incoming/outgoing functions that can be used. `MyPort` in the table header refers to `MyMessagePort` or `MyProcedurePort` in the example depending on the type of the port (message based or procedure based).

Table 1. Outgoing and incoming operations

		`MyPort::outgoing_`				`MyPort BASE::incoming_`			
		send	call	reply	raise	message	call	reply	exception
message type	in	○	○	○	○	●	○	○	○
	out	●	○	○	○	○	○	○	○
	inout	●	○	○	○	●	○	○	○
signature	in	○	○	●	●	○	●	○	○
	out	○	●	○	○	○	○	●	●
	inout	○	●	●	●	○	●	●	●

● supported

○ not supported

2.4.7. Additional Functions and Attributes

Any kind of attributes or member functions may be added to the Test Port. A file descriptor, which you communicate on, is almost always necessary. Names not interfering with the identifiers generated by the Compiler can be used in the header file (for example, the names containing one underscore character). Avoid using global variables because you may get confused when more than one instances of the Test Port run simultaneously. Any kind of software libraries may be used in the Test Port as well, but included foreign header files may cause name clashes between the library and the generated code.

In addition, the following `protected` attributes of ancestor classes are available:

Table 2. Protected attributes

Name	Type	Meaning
`is_started`	boolean	Indicates whether the Test Port is started.
`handler_installed`	boolean	Indicates whether the event handler is installed.
`port_name`	const char*	Contains the name of the Test Port instance. (NUL character terminated string)

Underscore characters are not duplicated in port name. In case of port array member instances the name string looks like this: `"Myport_array[5]"`.

2.5. Support of `address` Type

The special user-defined TTCN–3 type `address` can be used for addressing entities inside the SUT on ports mapped to the `system` component. Since the majority of Test Ports does not need TTCN–3 addressing and in order to keep the Test Port API backward compatible the support of `address` type is disabled by default. To enable addressing on a particular port type the extension attribute `"address"` must be added to the TTCN–3 port type definition. In addition to component references this extension will allow the usage `address` values or variables in the `to` or `from` clauses and `sender` redirects of port operations.

In order to use addressing, a type named `address` shall be defined in the same TTCN–3 module as the corresponding port type. Address types defined in other modules of the test suite do not affect the operation of the port type. It is possible to link several Test Ports that use different types for addressing SUT into the same executable test suite.

Test Ports that support SUT addressing have a slightly different API, which is considered when generating Test Port skeleton. This section summarizes only the differences from the normal API.

In the communication operations the test port author is responsible for handling the address information associated with the message or the operation. In case of an incoming message or operation the value of the received address will be stored in the port queue together with the received message or operation.

The generated code for the port skeleton of message based ports will be the same, except `outgoing_send` member function, which has an extra parameter pointing to an `ADDRESS` value. With the example given in Test Port Functions:

```
void outgoing_send(const INTEGER& send_par,
                   const ADDRESS *destination_address);
void outgoing_send(const CHARSTRING& send_par,
                   const ADDRESS *destination_address);
```

If an `address` value was specified in the `to` clause of the corresponding TTCN–3 `send` operation the second argument of `outgoing_send` points to that value. Otherwise it is set to the `NULL` pointer. The

Test Port code shall be prepared to handle both cases.

The outgoing operations of procedure based ports are also generated in the same way if the `address` extension is specified. These functions will also have an extra parameter. Based on our example, these will have the following form:

```
void outgoing_call(const MyExample::outProc_call& call_par,
                   const ADDRESS *destination_address);
void outgoing_call(const MyExample::inoutProc_call& call_par,
                   const ADDRESS *destination_address);
void outgoing_reply(const MyExample::inProc_reply& reply_par,
                    const ADDRESS *destination_address);
void outgoing_reply(const MyExample::inoutProc_reply& reply_par,
                    const ADDRESS *destination_address);
void outgoing_raise(const MyExample::inProc_exception& raise_exception,
                    const ADDRESS *destination_address);
void outgoing_raise(const MyExample::inoutProc_exception& raise_exception,
                    const ADDRESS *destination_address);
```

The other difference is in the `incoming_message` member function of class `MyMessagePort_BASE`, and in the incoming member functions of class `MyProcedurePort_BASE`. These have an extra parameter, which is a pointer to an `ADDRESS` value. The default value is set the NULL pointer. In our example of `MyMessagePort_BASE`:

```
void incoming_call(const MyExample::inProc_call& incoming_par,
                   const ADDRESS *sender_address = NULL);
void incoming_call(const MyExample::inoutProc_call& incoming_par,
                   const ADDRESS *sender_address = NULL);
void incoming_reply(const MyExample::outProc_reply& incoming_par,
                    const ADDRESS *sender_address = NULL);
void incoming_reply(const MyExample::inoutProc_reply& incoming_par,
                    const ADDRESS *sender_address = NULL);
void incoming_exception(const MyExample::outProc_exception& incoming_par,
                        const ADDRESS *sender_address = NULL);
void incoming_exception(const MyExample::inoutProc_exception& incoming_par,
                        const ADDRESS *sender_address = NULL);
```

If the event handler of the Test Port can determine the source address where the message or the operation is coming from, it shall pass a pointer to the incoming function, which points to a variable that stores the `address` value. The given address value is not modified by the run-time environment and a copy of it is created when the message or the operation is appended to the port queue. If the event handler is unable to determine the sender address the default `NULL` pointer shall be passed as second argument.

The address value stored in the port queue is used in `receive`, `trigger`, `getcall`, `getreply`, `catch` and `check` port operations: it is matched with the `from` clause and/or stored into the variable given in the `sender` redirect. If the receiving operation wants to use the address information of the first element in the port queue, but the Test Port has not supplied it a dynamic testcase error will occur.

2.6. Provider Port Types

Test Ports that belong to port types marked with `extension` attribute `"provider"` have a slightly different API. Such port types are used to realize dual-faced ports, the details of which can be found in section "Dual-faced ports" in the Programmer's Technical Reference.

The purpose of this API is to allow the re-use of the Test Port class with other port types marked with attribute `user` or with ports with translation capability (Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Configuration and Deployment Support). The user port types may have different lists of incoming and outgoing message types. The transformations between incoming and outgoing messages, which are specified entirely by the attribute of the user port type, are done independently of the Test Port. The Test Port needs to support the sending and reception of message types that are listed in the provider port type.

The provider port can be accessed through the port which maps to the port with provider attribute. The `get_provider_port()` is a member function of the PORT class:

```
PORT* get_provider_port();
```

This function is useful when a reference to the provider type is needed. It returns the provider port type for user ports and ports with translation capability. Otherwise returns NULL. The function causes dynamic testcase error when the port has more than one mapping, or the port has both mappings and connections. The function's return value must be manually cast to the correct provider port type.

This section summarizes only the differences from the normal Test Port API:

- The name of the Test Port class is suffixed with the string `_PROVIDER` (for example `MyMessagePort_PROVIDER` instead of `MyMessagePort`).
- The base class of the Test Port is class `PORT`, which is part of the Base Library. Please note that normal Test Ports are also derived from class PORT, but indirectly through an intermediate class with suffix `_BASE`.
- The member functions that handle incoming messages and procedure-based operations (that is `incoming_message`, `incoming_call`, `incoming_reply` and `incoming_exception`) must be defined in the header file as pure virtual functions. These functions will be implemented in various descendant classes differently.
- The Test Port header file must not include the generated header file of the corresponding TTCN–3 module. The common header file of the Base Library called TTCN3.hh shall be included instead. The source file of the Test Port may include any header file without restriction.
- The member functions of the Test Port may refer to C++ classes that are generated from user-defined message types and signatures. To avoid compilation failures the declarations of the referenced classes must be added to the beginning of the header file. At the moment the Test Port skeleton generator has a limitation that it cannot collect the class declarations from the port type, so they must be added manually. Please note that if a message type or signature is imported from another module the corresponding class declaration must be put into the

appropriate namespace.

The following example shows the generated Test Port skeleton of a provider port type.

Port type definition in TTCN–3 :

```
type port MyProviderPort mixed {
  inout MyMessage, MySignature;
} with { extension "provider" }
```

Header file `MyMessagePort.hh`:

```
// This Test Port skeleton header file was generated by the
// TTCN-3 Compiler of the TTCN-3 Test Executor version 1.7.pl0
// for Janos Zoltan Szabo (ejnosza@EG70E00202E46JR)
// on Wed Mar 7 18:14:33 2007


// Copyright (c) 2000-2026 Ericsson Telecom AB

// You may modify this file. Add your attributes and prototypes of your
// member functions here.


#ifndef MyProviderPort_HH
#define MyProviderPort_HH


#include <TTCN3.hh>

// Note: Header file MyExample.hh must not be included into this file!
// Class declarations were added manually

namespace MyOtherModule {
  // type MyMessageType was imported from MyOtherModule
  class MyMessageType;
}

namespace MyExample {

// signature MySignature was defined locally
class MySignature_call;
class MySignature_reply;
class MySignature_exception;
class MyProviderPort_PROVIDER : public PORT {
public:
  MyProviderPort_PROVIDER(const char *par_port_name = NULL);
  ~MyProviderPort_PROVIDER();
```

```
  void set_parameter(const char *parameter_name,
    const char *parameter_value);

  void Event_Handler(const fd_set *read_fds,
    const fd_set *write_fds, const fd_set *error_fds,
    double time_since_last_call);

protected:
  void user_map(const char *system_port);
  void user_unmap(const char *system_port);

  void user_start();
  void user_stop();

  void outgoing_send(const MyOtherModule::MyMessage& send_par);
  void outgoing_call(const MySignature_call& call_par);
  void outgoing_reply(const MySignature_reply& reply_par);
  void outgoing_raise(const MySignature_exception& raise_exception);
  virtual void incoming_message(
    const MyOtherModule::MyMessage& incoming_par) = 0;
  virtual void incoming_call(const MySignature_call& incoming_par) = 0;
  virtual void incoming_reply(const MySignature_reply& incoming_par) = 0;
  virtual void incoming_exception(
    const MySignature_exception& incoming_par) = 0;
};

} /* end of namespace */
```

Source file MyMessagePort.cc:

```
// This Test Port skeleton source file was generated by the
// TTCN-3 Compiler of the TTCN-3 Test Executor version 1.7.pl0
// for Janos Zoltan Szabo (ejnosza@EG70E00202E46JR)
// on Wed Mar 7 18:14:33 2007
// Copyright (c) 2000-2026 Ericsson Telecom AB
// You may modify this file. Complete the body of empty functions and
// add your member functions here.

#include "MyProviderPort.hh"
#include "MyExample.hh"

namespace MyExample {

MyProviderPort_PROVIDER::MyProviderPort_PROVIDER(const char *par_port_name)
  : PORT(par_port_name)
{
}

MyProviderPort_PROVIDER::~MyProviderPort_PROVIDER()
{
```

```
}

void MyProviderPort_PROVIDER::set_parameter(const char *parameter_name,
  const char *parameter_value)
{
}

void MyProviderPort_PROVIDER::Event_Handler(const fd_set *read_fds,
  const fd_set *write_fds, const fd_set *error_fds,
  double time_since_last_call)
{
}

void MyProviderPort_PROVIDER::user_map(const char *system_port)
{
}

void MyProviderPort_PROVIDER::user_unmap(const char *system_port)
{
}

void MyProviderPort_PROVIDER::user_start()
{
}

void MyProviderPort_PROVIDER::user_stop()
{
}

void MyProviderPort_PROVIDER::outgoing_send(
  const MyOtherModule::MyMessage& send_par)
{
}

void MyProviderPort_PROVIDER::outgoing_call(
  const MySignature_call& call_par)
{
}

void MyProviderPort_PROVIDER::outgoing_reply(
  const MySignature_reply& reply_par)
{
}

void MyProviderPort_PROVIDER::outgoing_raise(
  const MySignature_exception& raise_exception)
{
}

} /* end of namespace */
```

2.7. Tips and Tricks

The following sections deal with logging and error handling in Test Ports.

2.7.1. Logging

Test Ports may record important events in the Test Executor log during sending/receiving or encoding/decoding messages. Such log messages are also good for debugging fresh code.

The Test Port member functions may call the functions of class `TTCN_Logger`. These functions are detailed in Logging in Test Ports or External Functions.

If there are many points in the Test Port code that want to log something, it can be a good practice to write a common log function in the Test Port class. We show here an example function, which takes its arguments as the standard C function `printf` and forwards the message to the Test Executor's logger:

```
#include <stdarg.h>
// using in other member functions:
// log("The value of i: %d", i);
void MyPortType::log(const char *fmt, ...)
{
  // this flag can be a class member, which is configured through a
  // test port parameter
  if (logging_is_enabled) {
    va_list ap;
    va_start(ap, fmt);
    TTCN_Logger::begin_event(TTCN_DEBUG);
    TTCN_Logger::log_event("Example Test Port (%s): ", get_name());
    TTCN_Logger::log_event_va_list(fmt, ap);
    TTCN_Logger::end_event();
    va_end(ap);
  }
}
```

2.7.2. Error Handling

None of the Test Port member functions have return value like a status code. If a function returns normally, the run-time environment assumes that it has performed its task successfully. The handling of run-time errors is done in a special way, using C++ exceptions. This simplifies the program code because the return values do not have to be checked everywhere and dynamically created complex error messages can be used if necessary.

If any kind of fatal error is encountered anywhere in the Test Port, the following function should be called:

```
void TTCN_error(const char *err_msg, ⋯);
```

Its parameter should contain the description of the error in a NUL terminated string in the format of printf(3). You may pass further parameters to TTCN_error, if necessary. The function throws an exception, so it never returns. The exception is usually caught at the end of the test case or PTC function that is being executed. In case of error, the verdict of the component is set to error and the execution of the test case or PTC function terminates immediately.

The exception class is called TC_Error. For performance reasons this is a trivial (empty) class, that is, it does not contain the error message in a string. The error string is written into the log file by TTCN_error immediately. Such type of exception should never be caught or thrown directly. If you want to implement your own error handling and error recovery routines you had better use your own classes as exceptions.

If you write your own error reporting function you can add automatically the name of the port instance to all of your error messages. This makes the fault analysis for the end-users easier. In the following example the error message will occupy two consecutive lines in the log since we can pass only one format string to TTCN_error.

```
void MyPortType::error(const char *msg, ...)
{
  va_list ap;
  va_start(ap, msg);
  TTCN_Logger::begin_event(TTCN_ERROR);
  TTCN_Logger::log_event("Example Test Port (%s): ", get_name());
  TTCN_Logger::log_event_va_list(msg, ap);
  TTCN_Logger::end_event();
  va_end(ap);
  TTCN_error("Fatal error in Example Test Port %s (see above).",
    get_name());
}
```

There is another function for denoting warnings (that is, events that are not so critical) with the same parameter list as TTCN_error:

```
void TTCN_warning(const char *warning_msg, ⋯);
```

This function puts an entry in the executor's log with severity TTCN_WARNING. In contrast to TTCN_error, after logging the given message TTCN_warning returns and your test port can continue running.

2.8. Setting timestamps

In order to use the timestamp redirects (→ timestamp) described in chapter 5 of the TTCN-3 standard extension TTCN-3 Performance and Real Time Testing (ETSI ES 202 782 V1.3.1, [16]) the test port writer needs to add extra code to set the timestamps for the incoming and outgoing port operations of each port with the realtime clause.

2.8.1. Incoming operations

The timestamps of incoming port operations (`receive`, `trigger`, `getcall`, `getreply`, `catch` and `check`) need to be set when the incoming message or procedure is added to the queue.

The member functions `incoming_message`, `incoming_call`, `incoming_reply` and `incoming_exception` (which add the message/procedure to the queue) have an optional `float` parameter called `timestamp`, if the test port was declared with the `realtime` clause.

The value given to this parameter will be the one stored in the variable referenced in the timestamp redirect, if the operation has a timestamp redirect (otherwise the value is ignored).

It is recommended that this parameter be set to the current test system time, which can be queried with `TTCN_Runtime::now()`, or to a float variable that was set to the current test system time earier in the function.

Examples:

```
incoming_message(my_message, TTCN_Runtime::now());
```

```
FLOAT reply_time = TTCN_Runtime::now();

...

incoming_reply(my_reply, reply_time);
```

2.8.2. Outgoing operations

The timestamps of outgoing port operations (`send`, `call`, `reply`, `raise`) need to be set in the member functions `outgoing_send`, `outgoing_call`, `outgoing_reply` and `outgoing_raise`.

These functions have a `float` pointer parameter called `timestamp_redirect`, if the test port was declared with the `realtime` clause.

The value pointed to by this parameter will be the one stored in the variable referenced in the timestamp redirect, if the operation has a timestamp redirect.

If it does not have a timestamp redirect, then this pointer parameter will be `NULL`. Because of this, the parameter must always have a null pointer check before it is assigned a value.

It is recommended that the value pointed to by the parameter be set to the current test system time, which can be queried with `TTCN_Runtime::now()`.

Example:

```
if (timestamp_redirect != NULL) {
  *timestamp_redirect = TTCN_Runtime::now();
}
```

NOTE Because of this extra parameter, adding or removing the `realtime` clause from a port will cause already-written C++ code for the port to no longer compile. In these cases the new parameters must be manually added or removed from the mentioned functions, or the user-written code copied to newly-generated test port skeletons.

[1] The test equipment not necessarily requires a special hardware; it can even be a simple PC with an Ethernet interface.

[2] Test Port parameters have been introduced in version 1.1.pl3

[3] If the same parameter of the same port instance is specified several times in the configuration file, the function `set_parameter` will also be called several times.

[4] That is, the port has exactly one mapping and either the port has no connections or the message is explicitly addressed by a `send (···) to system` statement.

[5] In versions of Test Executor older than 1.1 the event handler function had no parameters. If you want to upgrade a test port developed for older versions, you should insert this formal parameter list to your event handler both in Test Port header and source file. Otherwise the compilation of Test Port will fail.

[6] Note that if the port has connections as well, the messages coming from other test components will also be inserted into the same queue independently from the event handler.

[7] In the example the signatures were defined in a different TTCN–3 module named MyExample, as a consequence all types defined in that module must be prefixed with the C++ namespace name of that module.

Chapter 3. External Classes

TITAN allows the declaration of external classes in TTCN-3 code, incomplete classes which the user can finish implementing, and extend further, in C++. To help with the implementation, the TTCN-3 compiler can generate external class skeletons with the command line option `-t`.

These skeletons contain the already implemented parts, as well as declarations and empty definitions for the external methods, of all external classes declared in TTCN-3 code. They also contain a constructor and destructor for each external class, even if these were not declared in TTCN-3.

Clarifications about the skeletons and implementation tips:

- One header (`.hh`) and one source file (`.cc`) is generated for each external class. The base names of these files (i.e. their names before the extensions) are equal to the names of their external class in TTCN-3. Please note that the name mapping rules described in Mapping of Names and Identifiers also apply to the file names.
- Both the generated header and source file can be extended with additional members and methods, however these new members and methods will not be visible in TTCN-3.
- The compiler will not regenerate the external class skeletons, if they already exist. Not even if the external class declaration in TTCN-3 changes. The compiler option `-f` (in addition to `-t`) can be used to force the regeneration of external class skeletons, however this will also overwrite any user code added to the previous skeletons.
- The function definitions generated for external methods are empty, aside from the code to reset `out` parameters, if there are any.
- The constructor's regular and external parameter lists in TTCN-3 are generated into one parameter list in C++ (with the regular parameters first, followed by the external parameters). The generated constructor's initializer list contains the call to the base constructor and the initialization of constant and template references. The constructor body contains the initializer code of any members with initial values in TTCN-3.
- A minimal implementation of the `log` function is also generated into each external class skeleton. This can be extended to display more information about the object while logging.
- The header file also contains a static inline function called `class_name`, which is used in error messages.
- The generated destructor automatically contains a try-catch block, to ensure that no exceptions are thrown from the destructor. All user code must be written inside the try-catch block.
- For more information about how to handle references of class types see Object References.

3.1. Example

Example.ttcn:

```
type external class ExternalClass {
  template integer t := ?;
  var charstring v;
  create(integer x) external (charstring y);
  public function f_ext(in integer x, out octetstring y) return boolean;
  function f_int(inout charstring p) { p := p & v; }
}
```

ExternalClass.hh:

```
// This external class skeleton header file was generated by the
// TTCN-3 Compiler of the TTCN-3 Test Executor version 8.2.0
// for ebotbar (ebotbar@ebotbar-VirtualBox) on Thu Jun  9 16:20:55 2022

// Copyright (c) 2000-2026 Ericsson Telecom AB

// You may modify this file. Add your attributes and prototypes of your
// member functions here.

#ifndef ExternalClass_HH
#define ExternalClass_HH

class ExternalClass :  public OBJECT {
public:
static const char* class_name() { return "ExternalClass"; }

protected:
const INTEGER_template& t;
INTEGER_template template_t;

protected:
CHARSTRING v;

public:
virtual BOOLEAN f__ext(const INTEGER& x, OCTETSTRING& y);

protected:
virtual void f__int(CHARSTRING& p);

public:
ExternalClass(const INTEGER& x, const CHARSTRING& y);

public:
virtual ~ExternalClass();

public:
virtual void log() const;
};

#endif
```

ExternalClass.cc:

```
// This external class skeleton source file was generated by the
// TTCN-3 Compiler of the TTCN-3 Test Executor version 8.2.0
// for ebotbar (ebotbar@ebotbar-VirtualBox) on Thu Jun  9 16:20:55 2022

// Copyright (c) 2000-2026 Ericsson Telecom AB
```

```
// You may modify this file. Complete the bodies of empty functions and
// add your member functions here.

#include "Example.hh"

namespace Example {

BOOLEAN ExternalClass::f__ext(const INTEGER& x, OCTETSTRING& y)
{
y.clean_up();

}

void ExternalClass::f__int(CHARSTRING& p)
{
/* Example.ttcn, line 8 */
{
CHARSTRING tmp_0;
tmp_0 = (p + v);
p = tmp_0;
}
}

ExternalClass::ExternalClass(const INTEGER& x, const CHARSTRING& y)
: OBJECT(), t(template_t)
{
/* Example.ttcn, line 4 */
template_t = ANY_VALUE;
}

ExternalClass::~ExternalClass()
{
try {

} catch (...) {
fprintf(stderr, "Unhandled exception or dynamic test case error in the destructor of
class `ExternalClass'. Terminating application.\n");
exit(EXIT_FAILURE);
}
}

void ExternalClass::log() const
{
TTCN_Logger::log_event_str("ExternalClass");
}


} /* end of namespace */
```

NOTE

The user is responsible for the handling of all constructor external parameters. External parameter references are not generated into the base constructor call automatically, if the base constructor also had external parameters. External parameter names may clash with that constructor's regular parameter names (since this is allowed by the compiler), and need to be adjusted manually.

Chapter 4. Logger Plug-ins

4.1. Implementing Logger Plug-ins

All logger plug-ins must implement the `ILoggerPlugin` interface class in `ILoggerPlugin.hh` in `${TTCN3_DIR}/include`. Each plug-in should provide some essential information on itself and should implement some basic functions:

The name `(name_, plugin_name())` of the plugin. To be able to reference the plugin (for example for configuration). Additional information about the plug-in `(help_, plugin_help())`.

The minimum API version number the plug-in is compatible with `(major_version_, major_version()`, `minor_version_, minor_version())`.

Each plug-in must have an initialization `(init())` and deinitialization `(fini())` routine, which are called at the begin and end of the plug-in's lifecycle. The same functionality can be implemented in the plug-in's constructor and destructor as well.

The plug-in could be asked, whether it's configured or not `(is_configured())`. For example the file is already opened, the database connection is set up etc. Depending on this information event buffering can be enabled or disabled.

One plug-in should provide `log2str()` functionality. The `is_log2str_capable()` function should be overridden to return true. At the moment it's not possible to change the default behavior and returning true will not have an effect except a warning.

The logger plug-ins receive the log events via the `log()` function. The details about event handling can be found in 3.3.

The generated, runtime specific (load-test or function-test) header file `TitanLoggerApi.hh` needs to be included by every logger plug-in depending on the runtime it is compiled for. These header files can be found in `${TTCN3_DIR}/include/{RT1/RT2}`. An example to handle these include files in a logger plug-in's code:

```
#ifndef TITAN_RUNTIME_2

#include ``RT1/TitanLoggerApi.hh''

#else

#include ``RT2/TitanLoggerApi.hh''

#endif
```

Unfortunately, the `dlopen()` API is a C API, not a C++ API, but each logger plug-in is a class, which needs to be instantiated. To resolve this, the logger plug-ins are always instantiated and destroyed through C factory functions. These functions are mandatory for all logger plug-ins and they must follow C-style linkage rules. Otherwise, the function names would be mangled by the C++ compiler,

using its own, implementation dependent mangling mechanism, and `dlsym()` and such functions would not be able to locate the correct symbol in the SOs of the logger plug-ins. These functions look pretty simple:

```
#ifdef __cplusplus
extern "C"
{
  ILoggerPlugin *create_plugin()
    { return new MyPlugin(); }
  void destroy_plugin(ILoggerPlugin *plugin)
    { delete plugin; plugin = NULL; }
}
#endif
```

4.2. Building Logger Plug-ins

The generated, runtime specific (load-test or function-test) header file `TitanLoggerApi.hh` needs to be included by every logger plug-in depending on the runtime it is compiled for. These header files can be found in `${TTCN3_DIR}/include/{RT1/RT2}` and this directory must be present (for example as part of `CPPFLAGS` in the `Makefile`) while compiling the logger plug-ins.

To make logger plug-ins dynamically loadable at runtime the logger plug-ins need to be built as shared libraries. Physically SOs `(.so)` on Unix and Linux platforms, DLLs `(.dll)` on Cygwin and Windows platforms. A HOWTO on building shared libraries can be found at David A. Wheeler, Program Library HOWTO. A quick summary:

All the sources of the logger plug-ins need to be compiled with `fPIC`, for example add `CXXFLAGS += -fPIC` into the `Makefile` or command line.

The linker should be instructed to create a shared library instead of an executable with the `shared` flag. `fPIC` is necessary here as well, for example add `LDFLAGS += -fPIC shared` in the `Makefile` or command line.

Another thing to keep in mind is that logger plug-ins need to be linked with the dynamically linked TITAN runtime libraries (for example `libttcn3-dynamic.so/libttcn3-parallel-dynamic.so` or `libttcn3-rt2-dynamic.so/libttcn3-rt2-parallel-dynamic.so`) instead of the static ones (for example `libttcn3.a/libttcn3-parallel.a` or `libttcn3-rt2.a/libttcn3-rt2-parallel.a`). So, if all possible combinations need to be supported by a logger plug-in, all of the four versions need to be built, additionally there are naming rules to simplify making a distinction between them:

- Single mode, load test runtime. File name must end with ".so".
- Single mode, function test runtime. File name must end with "-rt2.so".
- Parallel mode, load test runtime. File name must end with "-parallel.so".
- Parallel mode, function test runtime. File name must end with "-parallel-rt2.so".

The runtime library linked with a logger plug-in must be selected to match the runtime linked with the test executable that loads it: if the test executable is linked to `libttcn3-dynamic.so`, then any

logger plug-ins must also be linked to `libttcn3-dynamic.so` and not `libttcn3-parallel-dynamic.so` or `libttcn3-rt2-dynamic.so`. To ensure consistency, only a dynamic runtime library will load a logger plug-in (because a plug-in is always linked to a dynamic runtime library). If a non-dynamic runtime library is configured to load a logger plug-in, it will cause a runtime error.

Please note that linking a plug-in or any TTCN-3 project with the object files generated from the `TitanLoggerApi` or `TitanLoggerControl` internal modules and using the dynamic libraries of TITAN at the same time is not recommended and it can lead to various runtime errors.

4.3. Event Handling

The log events are distributed to all active logger plug-ins via a four-parameter callback function with the following signature:

```
void log(const TitanLoggerApi::TitanLogEvent& event, bool
  log_buffered, bool separate_file, bool use_emergency_mask);
```

The first parameter event is the event itself, the second parameter `log_buffered` indicates, whether the event is coming from an internal buffer or not, `separate_file` and `use_emergency_mask` are configuration options for emergency logging. The `use_emergency_mask` flag indicates that the given event is an emergency event and should be handled in a special way by the plug-ins, the `separate_file` flag indicates that all the emergency events should be handled separately (for example written into a separate file). For more details on emergency logging please check Programmer's Technical Reference. In this function, the plug-in can handle the log events individually depending on the event's type (that is, the alternative selected in the union `event.logEvent().choice()`).

`TitanLoggerApi::TitanLogEvent` is a generated type defined in TitanLoggerApi.xsd, which can be found in `${TTCN3_DIR}/include`. This file contains all the necessary type definitions a logger plug-in should be aware of. The corresponding header files generated from this XSD file can be found in `${TTCN3_DIR}/include/{RT1/RT2}`. The mapping between TTCN-3 types and C++ types is defined in Mapping TTCN–3 Data Types to C++ Constructs.

4.4. Execution

When a logger plug-in is compiled (the SO is ready) it should be configured in the configuration file. For details check Programmer's Technical Reference. Additionally, `LD_LIBRARY_PATH` should contain the directory of the plug-in and `${TTCN3_DIR}/lib` as well. If the runtime linker (the loader) is unable to find any of the given logger plug-ins an error will be given.

Chapter 5. Encoding and Decoding

This tool is equipped with several standard encoding/decoding mechanisms. A part of these functions reside in the core library, but the type-dependent part must be generated by the compiler. In order to reduce the code size and compilation time, the code generation for encoding functions (separately for different encoders) can be switched off if they are not needed. For details, see section "Command line syntax" in the Programmer's Technical Reference.

To make it easier to use the encoding features, a unified common API was developed. With help of this API the behaviour of the test executor in different error situations can be set during coding. There is also a common buffer class. The details of the above mentioned API as well as the specific features of the certain encoders are explained in the following sections.

5.1. The Common API

The common API for encoders consists of three main parts:

- A dummy class named `TTCN_EncDec` which encapsulates functions regarding error handling.
- A buffer class named `TTCN_Buffer` which is used by the encoders to put data in, decoders to get data from.
- The functions needed to encode and decode values.

5.1.1. TTCN_EncDec

`TTCN_EncDec` implements error handling functions.

Setting Error Behavior

There are lot of error situations during encoding and decoding. The coding functions can be told what to do if an error arises. To set the behaviour of test executor in a certain error situation the following function is to be invoked:

```
void TTCN_EncDec::set_error_behavior(error_type_t, error_behavior_t);
```

WARNING	As error_type_t and error_behavior_t are enums defined in TTCN_EncDec class, they have to prefixed with the class name and the scope operator (that is `TTCN_EncDec::`).

The possible values of `error_type_t` are detailed in the sections describing the different codings. Some common error types are shown in the table below:

Table 3. Common error types

ET_UNDEF	**Undefined/unknown error.**
ET_UNBOUND	Encoding of an unbound value.

ET_UNDEF	**Undefined/unknown error.**
ET_REPR	Representation error (for example, internal representation of integral numbers).
ET_ENC_ENUM	Encoding of an unknown enumerated value.
ET_DEC_ENUM	Decoding of an unknown enumerated value.
ET_INCOMPL_MSG	Decode error: incomplete message.
ET_INVAL MSG	Decode error: invalid message.
ET_CONSTRAINT	The value breaks some constraint.
ET_INTERNAL	Internal error. Error behaviour cannot be set for this.
ET_ALL	All error type. Usable only when setting error behaviour.
ET_NONE	No error.

The possible values of `error_behavior_t` are shown in the table below:

Table 4. Possible values of `error_behavior_t`

EB_DEFAULT	Sets the default error behaviour for the selected error type.
EB_ERROR	Raises an error if the selected error type occurs.
EB_WARNING	Gives a warning message but tries to continue the operation.
EB_IGNORE	Like warning but without the message.

Getting Error Behavior

There are two functions: one for getting the current setting and one for getting the default setting for a particular error situation.

```
error_behavior_t TTCN_EncDec::get_error_behavior(error_type_t);
error_behavior_t TTCN_EncDec::get_default_error_behavior(error_type_t);
```

The using of these functions are straightforward: giving a particular `error_type_t` the function returns the current or default `error_behavior_t` for that error situation, respectively.

Checking if an Error Occurred

The last coding-related error and its textual description can be retrieved anytime. Before using a coding function, it is advisable to clear the "last error". This can be achieved by the following method:

```
void TTCN_EncDec::clear_error();
```

After using some coding functions, it can be checked if an error occurred with this function:

```
error_type_t TTCN_EncDec::get_last_error_type();
```

This returns the last error, or `ET_NONE` if there was no error. The string representation of the error can be requested with the help of this:

```
const char* TTCN_EncDec::get_error_str();
```

WARNING	The above two functions do not clear the "last error" flag.

5.1.2. TTCN_Buffer

TTCN Buffer objects are used to store encoded values and to communicate with the coding functions. If encoding a value, the result will be put in a buffer, from which can be get. In the other hand, to decode a value, the encoded octet string must be put in a TTCN_Buffer object, and the decoding functions get their input from that.

```
void TTCN_Buffer::clear();
```

Resets the buffer, cleaning up its content, setting the pointers to the beginning of buffer.

```
void TTCN_Buffer::rewind();
```

Rewinds the buffer, that is, sets its reading pointer to the beginning of the buffer.

```
size_t TTCN_Buffer::get_pos() const;
```

Returns the (reading) position of the buffer.

```
void TTCN_Buffer::set_pos(size_t pos);
```

Sets the (reading) position to pos, or to the end of buffer, if `pos > get_len()`.

```
size_t TTCN_Buffer::get_len() const;
```

Returns the amount of bytes in the buffer.

```
const unsigned char* TTCN_Buffer::get_data() const;
```

Returns a pointer that points to the beginning of the buffer. You can read out `count` bytes beginning from this address, where `count` is the value returned by the `get_len()` member function.

```
size_t TTCN_Buffer::get_read_len() const;
```

Returns how many bytes are in the buffer to read.

```
const unsigned char* TTCN_Buffer::get_read_data() const;
```

Returns a pointer which points to the read position of data in the buffer. `count` bytes can be read out beginning from this address, where count is the value returned by the `get_read_len()` member function.

```
void TTCN_Buffer::put_c(const unsigned char c);
```

Appends the byte `c` to the end of buffer.

```
void TTCN_Buffer::put_s(const size_t len, const unsigned char *s);
```

Writes a string of bytes to the end of buffer, where len is the amount of bytes, `s` is a pointer to the data to be written.

```
void TTCN_Buffer::put_os(const OCTETSTRING& os);
```

Appends the content of the octet string to the buffer.

Sometimes it is useful to copy data directly into a buffer. In this case, the buffer must be told the maximum number of bytes to be written. So the buffer can resize its data area. This can be done with the following function:

```
void TTCN_Buffer::get_end(unsigned char*& end_ptr, size_t& end_len);
```

Parameter `end_len` is an in-out parameter: you tell how many bytes you want to write, and the returned value is equal to or greater than the requested. Parameter `end_ptr` is an out parameter. So up to `end_len` bytes can be written beginning from `end_ptr`. After writing also `increase_length()` must be called.

```
void TTCN_Buffer::increase_length(size_t count);
```

After writing bytes directly to the end of buffer using the pointer returned by `get_end()` method, the buffer must be told how many bytes have been written. This can be done by this function.

```
void TTCN_Buffer::cut();
```

Cuts (removes) the bytes between the beginning of the buffer and the read position. After calling this, the read position will be the beginning of buffer. As this function manipulates the internal data, pointers referencing to data inside the buffer will be invalid.

```
void TTCN_Buffer::cut_end();
```

Cuts (removes) the bytes between the read position and the end of the buffer. After calling this, the read position remains unchanged (that is, it will point to the end of the truncated buffer). As this function manipulates the internal data, pointers referencing to data inside the buffer will be invalid.

```
boolean TTCN_Buffer::contains_complete_TLV();
```

Returns `TRUE` if the buffer contains a complete TLV, otherwise it returns `FALSE`. Useful when decoding BER streams, and the data is coming in chunks. With the help of this, you can check before decoding whether the message is complete.

5.1.3. Invoking the Coding Functions

Every type class has members like these:

```
void encode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
    TTCN_EncDec::coding_t p_cod, ...) const;
void decode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
    TTCN_EncDec::coding_t p_cod, ...);
```

Parameter `p_td` is a special type descriptor. Each type has its own descriptor, which contains the name of the type, and a lot of information used by the different encoding mechanisms. The names of the descriptors come from the name of the types: the appropriate type descriptor for type `XXX` is `XXX_descr_`.

Parameter `p_buf` contains the encoded value. For details about using it, please consult the previous subsection.

Parameter `p_cod` is the desired coding mechanism. As `coding_t` is defined in `TTCN_EncDec`, its value must be prefixed with `TTCN_EncDec::`. For the time being, this parameter may have one of the following values:

- CT_BER - BER coding;
- CT_RAW - RAW coding;
- CT_TEXT - TEXT coding;
- CT_XER - XML coding;
- CT_JSON - JSON coding;
- CT_OER - OER coding;

- CT_PER - PER coding;

The optional ... parameter(s) are depending on the chosen coding.

5.2. BER

The encoding rules defined in Information TechnologyASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished can be used to encode and/or decode the values of ASN.1 types. There are three methods defined in the referenced document: BER, CER and DER (Basic, Canonical and Distinguished Encoding Rules). While the BER gives a lot of options to the sender (that is, to the encoder), the CER and DER select just one encoding from those allowed by the BER, eliminating all of the sender options. In other words, CER (and also DER) is a subset of BER. Any value encoded by CER or DER can be decoded using BER, but it is not true in the other direction.

In this section it is assumed that the reader has basic knowledge about BER, TLVs, tags, length forms and other items defined in Information TechnologyASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished.

This tool is capable of encoding values in CER or DER, and uses the BER while decoding[8]. The tags are handled quite separated from the types, giving extra freedom to the user when encoding only one component of a compound type. Let us suppose we have a large SEQUENCE with automatic tags (that is, context-specific implicit tags 1, 2, ...), the third component is `"[3] Other-sequence"`. Then we have the possibility to encode only this field using SEQUENCE-tag. (Implementation details and examples follow in next sections.)

5.2.1. Error Situations

In addition to error situations mentioned in The Common API, these can occur during BER-coding:

Table 5. BER-coding errors

ET_INCOMPL_ANY	Encoding of an ASN ANY value which does not contain a valid BER TLV.
ET_LEN_FORM	During decoding: the received message has a non-acceptable length form.
ET_TAG	During decoding: unexpected tag.
ET_SUPERFL	During decoding: superfluous part detected. This can be superfluous TLV at the end of a constructed TLV.
ET_EXTENSION	During decoding: there was something in the extension (for example: in ASN.1 ellipsis). This is not supported in the current version.
ET_DEC_DUPFLD	While decoding a SET: duplicated field (value for the given field already received).
ET_DEC_MISSFLD	While decoding a SET: missing field (value for the given field not received).

ET_DEC_OPENTYPE	Cannot decode an opentype (broken component relation constraint).
ET_DEC_UCSTR	While decoding a universal charstring: Malformed sequence.

5.2.2. API

The Application Programming Interface for ASN.1 type encoding and decoding is described in the following.

Encoding

```
void encode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
    TTCN_EncDec::coding_t p_cod, ...) const;
```

The parameter `p_cod` must be set to `TTCN_EncDec::CT_BER`. The variable argument list (...) must be set to one `int` argument, which is used to choose between CER and DER.

`BER_ENCODE_CER` = CER coding.

`BER_ENCODE_DER` = DER coding.

Decoding

```
void decode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
    TTCN_EncDec::coding_t p_cod, ...);
```

The parameter `p_cod` must be set to `TTCN_EncDec::CT_BER`. The variable argument list (...) must be set to one `int` argument, which determines which length forms are accepted.

- `BER_ACCEPT_SHORT`

  Short form.

- `BER_ACCEPT_LONG`

  Long form.

- `BER_ACCEPT_INDEFINITE`

  Indefinite form.

- `BER_ACCEPT_DEFINITE`

  Short and long form.

- `BER_ACCEPT_ALL`

  All form.

5.2.3. Example

Let us assume that we have an ASN.1 module named `MyASN` which contains a type named `ErrorReturn`, and we have a TTCN–3 module which imports this type. This module contains also two ports:

type port MyPort1 message

```
type port MyPort1 message
{
  out ErrorReturn;
  in octetstring;
}

type port MyPort2 message
{
  out octetstring;
  in ErrorReturn;
}
```

Then we can complete the port skeleton generated by the compiler:

```
void MyPort1::outgoing_send(const MyASN::ErrorReturn& send_par)
{
  TTCN_Buffer buf;
  send_par.encode(MyASN::ErrorReturn_descr_, buf,
                  TTCN_EncDec::CT_BER, BER_ENCODE_DER);
  OCTETSTRING encodeddata(buf.get_len(), buf.get_data());
  incoming_message(encodeddata);
}

void MyPort2::outgoing_send(const OCTETSTRING& send_par)
{
  TTCN_EncDec::set_error_behavior(TTCN_EncDec::ET_ALL,
                                  TTCN_EncDec::EB_WARNING);
  TTCN_Buffer buf;
  buf.put_os(send_par);
  MyASN::ErrorReturn pdu;
  pdu.decode(MyASN::ErrorReturn_descr_, buf, TTCN_EncDec::CT_BER,
             BER_ACCEPT_ALL);
  incoming_message(pdu);
}
```

5.3. RAW

You can use the encoding rules defined in the section "RAW encoder and decoder" in the Programmer's Technical Reference to encode and decode the following TTCN–3 types:

- boolean
- integer
- float
- bitstring
- octetstring
- charstring
- hexstring
- enumerated
- record
- set
- union
- record of
- set of

The compiler will produce code capable of RAW encoding/decoding for compound types if they have at least one `variant` attribute.
When a compound type is only used internally or it is never RAW encoded/decoded then the attribute `variant` has to be omitted.
When a type can be RAW encoded/decoded but with default specification then the empty variant specification can be used: `variant ""`.

5.3.1. Error Situations

Table 6. RAW-coding errors

ET_LEN_ERR	During encoding: Not enough length specified in FIELDLENGTH to encode the value. During decoding: the received message is shorter than expected.
ET_SIGN_ERR	Unsigned encoding of a negative number.
ET_FLOAT_NAN	Not a Number float value has been received.
ET_FLOAT_TR	The float value will be truncated during double to single precision conversion.

5.3.2. API

The C++ Application Programming Interface for RAW encoding and decoding is described in the following. It can be used for example in test port implementation, in external function implementation.

Encoding

```
void encode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
            TTCN_EncDec::coding_t p_cod, ...) const;
```

The parameter `p_cod` must be set to `TTCN_EncDec::CT_RAW`. The variable argument list (...) is ignored.

Decoding

```
void decode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
            TTCN_EncDec::coding_t p_cod, ...);
```

The parameter `p_cod` must be set to `TTCN_EncDec::CT_RAW`. The variable argument list (...) is ignored.

5.3.3. Example

Let us assume that we have a TTCN–3 module which contains a type named `ProtocolPdu`, and this module contains also two ports:

```
type port MyPort1 message
{
  out ProtocolPdu;
  in octetstring;
}

type port MyPort2 message
{
  out octetstring;
  in ProtocolPdu;
}
```

Then we can complete the port skeleton generated by the compiler as follows:

```
void MyPort1::outgoing_send(const ProtocolPdu& send_par)
{
  TTCN_Buffer buf;
  send_par.encode(ProtocolPdu_descr_, buf,
                  TTCN_EncDec::CT_RAW);
  OCTETSTRING encodeddata(buf.get_len(), buf.get_data());

  incoming_message(encodeddata);
}

void MyPort2::outgoing_send(const OCTETSTRING& send_par)
{
  TTCN_EncDec::set_error_behavior(TTCN_EncDec::ET_ALL,
                                  TTCN_EncDec::EB_WARNING);
  TTCN_Buffer buf;
  buf.put_os(send_par);
  ProtocolPdu pdu;
  pdu.decode(ProtocolPdu_descr_, buf, TTCN_EncDec::CT_RAW);

  incoming_message(pdu);
}
```

5.4. TEXT

You can use the encoding rules defined in the section "TEXT encoder, decoder" in the Programmer's Technical Reference to encode and decode the following TTCN–3 types:

- boolean
- integer
- charstring
- enumerated
- record
- set
- union
- record of
- set of

The compiler will produce code capable of TEXT encoding/decoding for compound types if they have at least one variant attribute or it is used within a compound type which has a TEXT attribute. If you need a compound type that is only used internally or it is never RAW encoded/decoded then you have to omit the variant attribute. If you need a type which can be TEXT encoded/decoded but with default specification then the empty variant specification can be used: `variant "TEXT_CODING()"`.

5.4.1. Error Situations

`ET_TOKEN_ERR` - The specified token is not found during decoding

5.4.2. API

The Application Programming Interface for TEXT encoding and decoding is described in the following.

Encoding

```
void encode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
            TTCN_EncDec::coding_t p_cod, ...) const;
```

The parameter `p_cod` must be set to `TTCN_EncDec::CT_TEXT`. The variable argument list (...) is ignored.

Decoding

```
void decode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
            TTCN_EncDec::coding_t p_cod, ...);
```

The parameter `p_cod` must be set to `TTCN_EncDec::CT_TEXT`. The variable argument list (...) is ignored.

5.4.3. Example

Let us assume that we have a TTCN–3 module which contains a type named `ProtocolPdu`, and this module contains also two ports:

```
type port MyPort1 message
{
  out ProtocolPdu;
  in charstring;
}

type port MyPort2 message
{
  out charstring;
  in ProtocolPdu;
}
```

Then we can complete the port skeleton generated by the compiler:

```
void MyPort1::outgoing_send(const ProtocolPdu& send_par)
{
  TTCN_Buffer buf;
  send_par.encode(ProtocolPdu_descr_, buf,
                  TTCN_EncDec::CT_TEXT);
  CHARSTRING encodeddata(buf.get_len(), buf.get_data());

  incoming_message(encodeddata);
}

void MyPort2::outgoing_send(const CHARSTRING& send_par)
{
  TTCN_EncDec::set_error_behavior(TTCN_EncDec::ET_ALL,
                                  TTCN_EncDec::EB_WARNING);
  TTCN_Buffer buf;
  buf.put_cs(send_par);
  ProtocolPdu pdu;
  pdu.decode(ProtocolPdu_descr_, buf, TTCN_EncDec::CT_TEXT);

  incoming_message(pdu);
}
```

5.5. XML Encoding (XER)

The encoding rules defined by Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 9: Using XML Schema with TTCN–3 European can be used to encode and/or decode values of ASN.1 and TTCN-3 types. This tool is capable of encoding and decoding Basic XER (BXER), Canonical XER (CXER) and Extended XER (EXER). Values of all ASN.1 types can be encoded, but only BXER and CXER are available for them because parsing XML Encoding Instructions in ASN.1 files is not implemented.

The following built-in TTCN-3 types can be encoded in XML:

- boolean
- integer
- float
- bitstring
- octetstring
- hexstring
- objid
- charstring
- universal charstring
- verdicttype

The following user-defined types can be encoded in XML:

- enumerated types
- record, set and union types, if all components can be encoded.
- record of and set of types, if the type of the element can be encoded.

The encoder and the decoder are working with XML data encoded in UTF-8 (described in UTF-8, a transformation format of ISO 10646), stored in an object of type `TTCN_buffer`. Although the contents of this object can be retrieved (using the overloads of the get_string function) as an instance of `OCTETSTRING`, `CHARSTRING` or `UNIVERSAL_CHARSTRING`, it is recommended to use only the `OCTETSTRING` representation. `CHARSTRING` is not recommended, because UTF-8 is an 8-bit encoding so the buffer may contain bytes with values over 127, which are not valid characters for a TTCN-3 `charstring` (which is implemented by `CHARSTRING`, see Charstring). `UNIVERSAL_CHARSTRING` must not be used because its internal representation is not UTF-8.

5.5.1. Error Situations

In addition to error situations mentioned in The Common API, the following can occur during XMLcoding:

Table 7. XER coding errors

ET_TAG	Incorrect (unexpected) XML tag found during decoding

5.5.2. API

The Application Programming Interface for XML encoding and decoding is described in the following.

Encoding

```
void encode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
    TTCN_EncDec::coding_t p_cod, ...) const;
```

The parameter `p_cod` must be set to `TTCN_EncDec::CT_XER`. The variable argument list (...) must be set to one `int` argument, which is used to choose between BXER, CXER and EXER:

`XER_BASIC` = Basic XER (BXER)

`XER_CANONICAL` = Canonical XER (CXER)

`XER_EXTENDED` = Extended XER (EXER)

Decoding

```
void decode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
    TTCN_EncDec::coding_t p_cod, ...);
```

The parameter `p_cod` must be set to `TTCN_EncDec::CT_XER`. The variable argument list (...) must be set to one `int` argument, which is used to choose between BXER, CXER and EXER:

`XER_BASIC` = Basic XER (BXER)

`XER_CANONICAL` = Canonical XER (CXER)

`XER_EXTENDED` = Extended XER (EXER)

5.5.3. Example

Let us assume that we have a TTCN–3 module which contains a type named `ProtocolPdu`, and this module contains also two ports:

```
void MyPort1::outgoing_send(const ProtocolPdu& send_par)
{
  TTCN_Buffer buf;
  send_par.encode(ProtocolPdu_descr_, buf,
                  TTCN_EncDec::CT_XER, XER_EXTENDED);
  OCTETSTRING encodeddata(buf.get_len(), buf.get_data());

  incoming_message(encodeddata);
}

void MyPort2::outgoing_send(const OCTETSTRING& send_par)
{
  TTCN_EncDec::set_error_behavior(TTCN_EncDec::ET_ALL,
                                  TTCN_EncDec::EB_WARNING);
  TTCN_Buffer buf;
  buf.put_os(send_par);
  ProtocolPdu pdu;
  pdu.decode(ProtocolPdu_descr_, buf, TTCN_EncDec::CT_XER, XER_EXTENDED);

  incoming_message(pdu);
}
```

5.6. JSON

The encoding rules defined in the section "JSON Encoder and Decoder" of the Programmer's Technical Reference can be used to encode and decode the following TTCN–3 types:

- anytype
- array
- bitstring
- boolean
- charstring
- enumerated

- float
- hexstring
- integer
- objid
- octetstring
- record`, set
- record of`, set of
- union
- universal charstring
- verdicttype

The rules also apply to the following ASN.1 types (if imported to a TTCN-3 module):

- ANY
- BIT STRING
- BOOLEAN
- BMPString
- CHOICE, open type (in instances of parameterized types)
- ENUMERATED
- GeneralString
- GraphicString
- IA5String
- INTEGER
- NULL
- NumericString
- OBJECT IDENTIFIER
- OCTET STRING
- PrintableString
- RELATIVE`-OID
- SEQUENCE, SET
- SEQUENCE OF, SET OF
- TeletexString
- UniversalString
- UTF8String
- VideotexString
- VisibleString

The compiler will produce code capable of JSON encoding/decoding for compound types if they have at least one JSON variant attribute or the `encode "JSON"` attribute (and, for compound types, all fields and elements of compound types also have a JSON variant attribute or the `encode "JSON"` attribute).

The encoder and the decoder work with JSON data encoded in UTF-8 (described in UTF-8, a transformation format of ISO 10646), stored in an object of type `TTCN_buffer`. Although the contents of this object can be retrieved (using the overloads of the `get_string` function) as an instance of `OCTETSTRING`, `CHARSTRING` or `UNIVERSAL_CHARSTRING`, it is recommended to use only the `OCTETSTRING` representation. `CHARSTRING` is not recommended, because UTF-8 is an 8-bit encoding so the buffer may contain bytes with values over 127, which are not valid characters for a TTCN-3 `charstring` (which is implemented by `CHARSTRING`, see Charstring). `UNIVERSAL_CHARSTRING` must not be used because its internal representation is not UTF-8.

5.6.1. Error Situations

There are no extra error situations apart from the ones in The Common API.

5.6.2. API

The Application Programming Interface for JSON encoding and decoding is described in the following.

Encoding

```
void encode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
            TTCN_EncDec::coding_t p_cod, ...) const;
```

The parameter `p_cod` must be set to `TTCN_EncDec::CT_JSON`. The variable argument list (...) must be set to one `int` argument, which indicates how the JSON language elements are printed:

`0` = compact printing (i.e. no extra whitespaces are added between JSON language elements),

`1` = pretty printing (i.e. extra whitespaces are added between JSON language elements).

Decoding

```
void decode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
            TTCN_EncDec::coding_t p_cod, ...);
```

The parameter `p_cod` must be set to `TTCN_EncDec::CT_JSON`. The variable argument list (...) is ignored.

5.6.3. Example

Let us assume that we have a TTCN–3 module which contains a type named `ProtocolPdu`, and this module also contains two ports:

```
type port MyPort1 message
{
  out ProtocolPdu;
  in octetstring;
}

type port MyPort2 message
{
  out octetstring;
  in ProtocolPdu;
}
```

Then we can complete the port skeleton generated by the compiler:

```
void MyPort1::outgoing_send(const ProtocolPdu& send_par)
{
  TTCN_Buffer buf;
  send_par.encode(ProtocolPdu_descr_, buf,
                  TTCN_EncDec::CT_JSON);
  OCTETSTRING encodeddata(buf.get_len(), buf.get_data());

  incoming_message(encodeddata);
}

void MyPort2::outgoing_send(const OCTETSTRING& send_par)
{
  TTCN_EncDec::set_error_behavior(TTCN_EncDec::ET_ALL,
                                  TTCN_EncDec::EB_WARNING);
  TTCN_Buffer buf;
  buf.put_os(send_par);
  ProtocolPdu pdu;
  pdu.decode(ProtocolPdu_descr_, buf, TTCN_EncDec::CT_JSON);

  incoming_message(pdu);
}
```

5.7. OER

The encoding rules defined in the section "OER Encoder and Decoder" of the Programmer's Technical Reference can be used to encode and/or decode the values of ASN.1 types.

5.7.1. Error Situations

There are no extra error situations apart from the ones in The Common API.

5.7.2. API

The Application Programming Interface for ASN.1 type encoding and decoding is described in the following.

Encoding

```
void encode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
    TTCN_EncDec::coding_t p_cod, ...) const;
```

The parameter `p_cod` must be set to `TTCN_EncDec::CT_OER`. The variable argument list (...) is ignored.

Decoding

```
void decode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
    TTCN_EncDec::coding_t p_cod, ...);
```

The parameter `p_cod` must be set to `TTCN_EncDec::CT_OER`. The variable argument list (...) is ignored.

5.7.3. Example

Let us assume that we have an ASN.1 module named `MyASN` which contains a type named `ErrorReturn`, and we have a TTCN–3 module which imports this type. This module also contains two ports:

```
type port MyPort1 message
{
  out ErrorReturn;
  in octetstring;
}

type port MyPort2 message
{
  out octetstring;
  in ErrorReturn;
}
```

Then we can complete the port skeleton generated by the compiler:

```
void MyPort1::outgoing_send(const MyASN::ErrorReturn& send_par)
{
  TTCN_Buffer buf;
  send_par.encode(MyASN::ErrorReturn_descr_, buf,
                  TTCN_EncDec::CT_OER);
  OCTETSTRING encodeddata(buf.get_len(), buf.get_data());
  incoming_message(encodeddata);
}

void MyPort2::outgoing_send(const OCTETSTRING& send_par)
{
  TTCN_EncDec::set_error_behavior(TTCN_EncDec::ET_ALL,
                                  TTCN_EncDec::EB_WARNING);
  TTCN_Buffer buf;
  buf.put_os(send_par);
  MyASN::ErrorReturn pdu;
  pdu.decode(MyASN::ErrorReturn_descr_, buf, TTCN_EncDec::CT_OER);
  incoming_message(pdu);
}
```

5.8. PER

The encoding rules defined in the section "PER Encoder and Decoder" of the Programmer's Technical Reference can be used to encode and/or decode the values of ASN.1 types.

5.8.1. Error Situations

There are no extra error situations apart from the ones in The Common API.

5.8.2. API

The Application Programming Interface for PER encoding and decoding is described in the following.

Encoding

```
void encode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
    TTCN_EncDec::coding_t p_coding, ...) const;
```

The parameter `p_coding` must be set to `TTCN_EncDec::CT_PER`.

The variable argument list (...) must be set to one `int` argument. The following flags are available for convenience:

- basic, unaligned PER: `PER_UNALIGNED`, `PER_BASIC`;
- aligned PER: `PER_ALIGNED`;

- canonical PER: `PER_CANONICAL`.

Decoding

```
void decode(const TTCN_Typedescriptor_t& p_td, TTCN_Buffer& p_buf,
    TTCN_EncDec::coding_t p_coding, ...);
```

The parameter `p_coding` must be set to `TTCN_EncDec::CT_PER`.

The variable argument list (...) must be set to one `int` argument. The following flags are available for convenience:

- basic, unaligned PER: `PER_UNALIGNED`, `PER_BASIC`;
- aligned PER: `PER_ALIGNED`;
- canonical PER: `PER_CANONICAL`.

5.8.3. Example

Let us assume that we have an ASN.1 module named `MyASN` which contains a type named `ErrorReturn`, and we have a TTCN–3 module which imports this type. This module also contains two ports:

```
type port MyPort1 message
{
  out ErrorReturn;
  in octetstring;
}

type port MyPort2 message
{
  out octetstring;
  in ErrorReturn;
}
```

Then we can complete the port skeleton generated by the compiler:

```
void MyPort1::outgoing_send(const MyASN::ErrorReturn& send_par)
{
  TTCN_Buffer buf;
  send_par.encode(MyASN::ErrorReturn_descr_, buf,
                  TTCN_EncDec::CT_PER, PER_ALIGNED);
  OCTETSTRING encodeddata(buf.get_len(), buf.get_data());
  incoming_message(encodeddata);
}

void MyPort2::outgoing_send(const OCTETSTRING& send_par)
{
  TTCN_EncDec::set_error_behavior(TTCN_EncDec::ET_ALL,
                                  TTCN_EncDec::EB_WARNING);
  TTCN_Buffer buf;
  buf.put_os(send_par);
  MyASN::ErrorReturn pdu;
  pdu.decode(MyASN::ErrorReturn_descr_, buf,
             TTCN_EncDec::CT_PER, PER_ALIGNED);
  incoming_message(pdu);
}
```

[8] Though the decoder can be forced to accept only certain length forms (short, long, indefinite or any combination of these.

Chapter 6. Mapping TTCN–3 Data Types to C++ Constructs

The TTCN–3 language elements of the test suite are individually mapped into more or less equivalent C++ constructs. The data types are mapped to C++ classes, the test cases become C++ functions, and so on. In order to write a Test Port, it is inevitable to be familiar with the internal representation format of TTCN–3 data types and values. This section gives an overview about the data types and their equivalent C++ constructs.

6.1. Mapping of Names and Identifiers

In order to identify the TTCN–3 language elements in the generated C++ program properly, the names of test suite are translated to C++ identifiers according to the following simple rules.

If the TTCN–3 identifier does not contain any underscore (_) character, its equivalent C++ identifier will be the same. For example, the TTCN–3 variable `MyVar` will be translated to a C++ variable called `MyVar`.

If the TTCN–3 identifier contains one or more underscore characters, each underscore character will be duplicated in the C++ identifier. So the TTCN–3 identifier `My_Long_Name` will be mapped to a C++ identifier called `My__Long__Name`.

The idea behind this name mapping is that we may freely use the C++ identifiers containing one underscore character in the generated code and in the Test Ports as well. Otherwise name clashes can always happen because the name space of TTCN–3 and C++ is identical. Furthermore, the generated C++ language elements fulfill the condition that the scope of a translated C++ identifier is identical as the scope of the original TTCN–3 identifier.

The identifiers that are keywords of C or C++ but not keywords in TTCN–3 are mapped to themselves, but a single underscore character is appended at the end (for example `typedef` becomes `typedef_`). The same rule applies to the all-uppercase identifiers that are used in the Base Library: identifier `INTEGER` in TTCN–3 becomes `INTEGER_` in C++, `TRUE` [9] is mapped to `TRUE_`, etc.

Here is the complete list (in alphabetical order) of the identifiers that are handled in such special way:asm, auto, bitand, bitor, bool, break, case, class, compl, continue, delete, double, enum, explicit, export, friend, inline, int, ischosen, long, main, mutable, namespace, new, operator, private, protected, public, register, short, signed, static, stderr, stdin, stdout, struct, switch, this, throw, try, typedef, typeid, typename, unsigned, using, virtual, void, volatile, ADDRESS, BITSTRING, BOOLEAN, CHAR, CHARSTRING, COMPONENT, DEFAULT, ERROR, FAIL, FALSE, FLOAT, HEXSTRING, INCONC, INTEGER, NONE, OBJID, OCTETSTRING, PASS, PORT, TIMER, TRUE, VERDICTTYPE.

The identifiers that are the names of common preprocessor macros of the C library (such as `putchar`, `errno` or `NULL`) should be avoided in TTCN–3 modules. The name clashes with macros can cause mysterious compilation error messages.

Note that these name mapping rules apply to **all** TTCN–3 identifiers, including module, Test Port, type, field, variable and function names.

WARNING	By default, from version 1.2.pl0 the compiler does NOT duplicate the underscores in output file names and file references (for example when handling imports).

6.2. Namespaces

The compiler generates a C++ namespace for every TTCN–3 and ASN.1 module. All C++ definitions that belong to the module (including Test Port classes and external functions) are placed in that namespace. The name of the namespace is derived from the module identifier according to the rules described in Mapping of Names and Identifiers.

The definitions of the TTCN–3 Base Library do not use any namespace.

When accessing a C++ entity that belongs to a different module than the referring Test Port or external function is in the reference has to be prefixed with the namespace of the referenced module. For example, to access the C++ class that realizes type `MyType` defined in module `MyExample1` from a Test Port that belongs to module `MyExample2` the reference shall be written as `MyExample1::MyType`.

6.3. Predefined TTCN–3 Data Types

There are some basic data types in TTCN–3 that have no equivalent data types in language C/C++ (for example bitstring, verdicttype). Other types have C++ equivalent, but the TTCN–3 executor must know whether a variable has a valid value or not because sending an unbound value must result in a dynamic test case error. Thus, in the TTCN–3 Base Library all basic data types of TTCN–3 were implemented as C++ classes. This section describes the member functions of these classes.

6.3.1. `Integer`

The TTCN–3 type `integer` is implemented in class `INTEGER`.
The class `INTEGER` has the following public member functions:

Table 8. Public member functions of the class `INTEGER`

Member functions		Notes
Constructors	`INTEGER()`	Initializes to unbound value.
	`INTEGER(int)`	Initializes to a given value.
	`INTEGER(const INTEGER&)`	Copy constructor.
	`explicit INTEGER(const char *)`	Initializes with the (NUL terminated) string representation of an integer.
	`explicit INTEGER(BIGNUM *other_value)`	Initializes with a BIGNUM object.
Destructor	`~INTEGER()`	

Assignment operators	`INTEGER()`	Initializes to unbound value.
	`INTEGER()`	Initializes to unbound value.
Comparison operators	`boolean operator==(int) const`	Returns TRUE if equals
	`boolean operator==(const INTEGER&) const`	and FALSE otherwise.
	`boolean operator!=(int) const`	
	`boolean operator!=(const INTEGER&) const`	
	`boolean operator<(int) const`	
	`boolean operator<(const INTEGER&) const`	
	`boolean operator⇐(int) const`	
	`boolean operator⇐(const INTEGER&) const`	
	`boolean operator>(int) const`	
	`boolean operator>(const INTEGER&) const`	
	`boolean operator>=(int) const`	
	`boolean operator>=(const INTEGER&) const`	
Arithmetic operators	`INTEGER operator+() const`	Unary plus.
	`INTEGER operator-() const`	Unary minus.
	`INTEGER operator+(int) const`	Addition.
	`INTEGER operator+(const INTEGER&) const`	
	`INTEGER operator-(int) const`	Subtraction.
	`INTEGER operator-(const INTEGER&) const`	
	`INTEGER operator*(int) const`	Multiplication.
	`INTEGER operator*(const INTEGER&) const`	
	`INTEGER operator/(int) const`	Integer division.
	`INTEGER operator/(const INTEGER&) const`	
	`INTEGER& operator++()`	Incrementation (prefix).
	`INTEGER& operator—()`	Decrementation (prefix).
Casting operator	`operator int() const`	Returns the value.

Other member functions	`void log() const`	Puts the value into log.
	`boolean is_bound() const`	Returns whether the value is bound.
	`void clean_up()`	Deletes the value, setting it to unbound.
	`long long int get_long_long_val() const`	Returns the value as a long long `int`.
	`void set_long_long_val(long long int)`	Sets the given long long `int` value.

The comparison, arithmetic and shifting operators are also available as global functions for that case when the left side is `int` and the right side is `INTEGER`. Using the value of an unbound variable for anything will cause dynamic test case error.

The casting operator `int()` is applicable only to `INTEGER` objects holding a signed value with at most 31 useful bits, since in C/C++ the native `int` type is 32-bit large including the sign bit. Casting an `INTEGER` object holding a bigger (for example a 32-bit unsigned) value will result in run-time error.

Please note that if the value stored in an `INTEGER` object is too big (that is, it cannot be represented as a `long long int`) the value returned by `get_long_long_val()` will contain only the lowest `sizeof(long long int)` bytes of the original value. Another way to obtain a value of a number having more useful bits than 31 is to convert the INTEGER object to its string representation using the `int2str()` predefined function. Then the string value can be converted to any native integer type using the `sscanf()` library function or such. The following example demonstrates a common scenario:

```
unsigned int get_unsigned_int_val(const INTEGER& other_value)
{
  unsigned int ret_val = 0;
  sscanf((const char *)int2str(), "%u", &ret_val);
  return ret_val;
}
```

In addition, the following global functions are available for modulo division. These functions return the result of `mod` and `rem` operations according to TTCN–3 semantics.

```
INTEGER mod(const INTEGER& left_operand, const INTEGER& right_operand);
INTEGER mod(const INTEGER& left_operand, int right_operand);
INTEGER mod(int left_operand, const INTEGER& right_operand);
INTEGER mod(int left_operand, int right_operand);

INTEGER rem(const INTEGER& left_operand, const INTEGER& right_operand);
INTEGER rem(const INTEGER& left_operand, int right_operand);
INTEGER rem(int left_operand, const INTEGER& right_operand);
INTEGER rem(int left_operand, int right_operand);
```

Other operators (global functions):

```
INTEGER operator+(int int_value, const INTEGER& other_value);  // Add
INTEGER operator-(int int_value, const INTEGER& other_value);  // Subtract
INTEGER operator*(int int_value, const INTEGER& other_value);  // Multiply
INTEGER operator/(int int_value, const INTEGER& other_value);  // Divide
boolean operator==(int int_value, const INTEGER& other_value); // Equal
boolean operator!=(int int_value, const INTEGER& other_value); // Not equal
boolean operator<(int int_value, const INTEGER& other_value);  // Less than
boolean operator>(int int_value, const INTEGER& other_value);  // More than
```

6.3.2. `Float`

The TTCN–3 type `float` is implemented in class `FLOAT`.
The class `FLOAT` has the following public member functions:

Table 9. Public member functions of the class `FLOAT`

Member functions		**Notes**
Constructors	`FLOAT()`	Initializes to unbound value.
	`FLOAT(double)`	Initializes to a given value.
	`FLOAT(const FLOAT&)`	Copy constructor.
Destructor	`~FLOAT()`	
Assignment operators	`FLOAT& operator=(double)`	Assigns the given value
	`FLOAT& operator=(const FLOAT&)`	and sets the bound flag.

Comparison operators	`boolean operator==(double) const`	Returns TRUE if equals
	`boolean operator==(const FLOAT&) const`	and FALSE otherwise.
	`boolean operator!=(double) const`	
	`boolean operator!=(const FLOAT&) const`	
	`boolean operator<(double) const`	
	`boolean operator<(const FLOAT&) const`	
	`boolean operator⇐(double) const`	
	`boolean operator⇐(const FLOAT&) const`	
	`boolean operator>(double) const`	
	`boolean operator>(const FLOAT&) const`	
	`boolean operator>=(double) const`	
	`boolean operator>=(const FLOAT&) const`	
Arithmetic operators	`double operator+() const`	Unary plus.
	`double operator-() const`	Unary minus.
	`double operator+(double) const`	Addition.
	`double operator+(const FLOAT&) const`	
	`double operator-(double) const`	Subtraction.
	`double operator-(const FLOAT&) const`	
	`double operator*(double) const`	Multiplication.
	`double operator*(const FLOAT&) const`	
	`double operator/(double) const`	Division.
	`double operator/(const FLOAT&) const`	
Casting operator	`operator double() const`	Returns the value.

Other member functions	`void log() const`	Puts the value into log, either in exponential or decimal dot notation.
	`boolean is_bound() const`	Returns whether the value is bound.
	`void clean_up()`	Deletes the value, setting it to unbound.

The comparison and arithmetic operators are also available as global functions for that case when the left side is `double` and the right side is `FLOAT`. Using the value of an unbound variable for anything will cause dynamic test case error.

Other operators (global functions):

```
FLOAT operator+(double double_value, const FLOAT& other_value);    // Add
FLOAT operator-(double double_value, const FLOAT& other_value);    // Subtract
FLOAT operator*(double double_value, const FLOAT& other_value);    // Multiply
FLOAT operator/(double double_value, const FLOAT& other_value);    // Divide
boolean operator==(double double_value, const FLOAT& other_value); // Equal
boolean operator!=(double double_value, const FLOAT& other_value); // Not equal
boolean operator<(double double_value, const FLOAT& other_value);  // Less than
boolean operator>(double double_value, const FLOAT& other_value);  // More than
```

6.3.3. `Boolean`

The TTCN–3 type `boolean` is implemented in class `BOOLEAN`.We have introduced an ancillary C enumerated type called `boolean` to set and get values. It may have two predefined values: `TRUE` or `FALSE`. You may use `boolean` values in C conditions since `FALSE` equals to zero and `TRUE` is not zero.
The class `BOOLEAN` has the following public member functions:

Table 10. Public member functions of the class `BOOLEAN`

Member functions		**Notes**
Constructors	`BOOLEAN()`	Initializes to unbound value.
	`BOOLEAN(boolean)`	Initializes to a given value.
	`BOOLEAN(const BOOLEAN&)`	Copy constructor.
Destructor	`~BOOLEAN()`	
Assignment operators	`BOOLEAN& operator=(boolean)`	Assigns the given value
	`BOOLEAN& operator=(const BOOLEAN&)`	and sets the bound flag.

Comparison operators	`boolean operator==(boolean) const`	Returns TRUE if equals
	`boolean operator==(const BOOLEAN&) const`	and FALSE otherwise.
	`boolean operator!=(boolean) const`	Same as XOR.
	`boolean operator!=(const BOOLEAN&) const`	
Logical operators	`boolean operator!() const`	Negation (NOT).
	`boolean operator&&(boolean) const`	Logical AND.
	`boolean operator&&(const BOOLEAN&) const`	
	`boolean operator|(boolean) const`	Logical OR.
	`boolean operator|(const BOOLEAN&) const`	
	`boolean operator^(boolean) const`	Exclusive or (XOR).
	`boolean operator^(const BOOLEAN&) const`	
Casting operator	`operator boolean() const`	Returns the value.
Other member functions	`void log() const`	Puts the value into log. Like “TRUE” or “FALSE”.
	`boolean is_bound() const`	Returns whether the value is bound
	`void clean_up()`	Deletes the value, setting it to unbound.

The comparison and logical operators are also available as global functions for that case when the left side is `boolean` and the right side is `BOOLEAN`. Using the value of an unbound variable for anything will cause dynamic test case error.

Other operators (global functions):

```
BOOLEAN operator&&(boolean bool_value, const BOOLEAN& other_value); // And
BOOLEAN operator^(boolean bool_value, const BOOLEAN& other_value);  // Not
BOOLEAN operator||(boolean bool_value, const BOOLEAN& other_value); // Or
boolean operator==(boolean bool_value, const BOOLEAN& other_value); // Equal
boolean operator!=(boolean bool_value, const BOOLEAN& other_value);// Not equal
```

6.3.4. `Verdicttype`

The TTCN–3 type `verdicttype` is implemented in class `VERDICTTYPE`. We have introduced an ancillary C enumerated type called `verdicttype` to set and get values. It may have five predefined values:

NONE, PASS, INCONC, FAIL or ERROR. The order of these values is NONE < PASS < INCONC < FAIL < ERROR. The class VERDICTTYPE has the following public member functions:

Table 11. Public member functions of the class VERDICTTYPE

Member functions		Notes
Constructors	VERDICTTYPE()	Initializes to unbound value.
	VERDICTTYPE(verdicttype)	Initializes to a given value.
	VERDICTTYPE(const VERDICTTYPE&)	Copy constructor.
Destructor	~VERDICTTYPE()	
Assignment operators	VERDICTTYPE& operator=(verdicttype)	Assigns the given value
	VERDICTTYPE& operator= (const VERDICTTYPE&)	and sets the bound flag.
Comparison operators	boolean operator==(verdicttype) const	Returns TRUE if equals
	boolean operator==(const VERDICTTYPE&) const	and FALSE otherwise.
	boolean operator!=(verdicttype) const	
	boolean operator!=(const VERDICTTYPE&) const	
Casting operator	operator verdicttype() const	Returns the value.
Other member functions	Puts the value into log.	void log() const
	Puts the value into log.	Like “pass” or “fail”.
	boolean is_bound() const	Returns whether the value is bound.

The comparison operators are also available as global functions for that case when the left side is verdicttype and the right side is VERDICTTYPE. Using the value of an unbound VERDICTTYPE variable for anything will cause dynamic test case error.

From version 1.2.pl0 there are the following three static member functions in class TTCN_Runtime defined in the Base Library for getting or modifying the local verdict of the current test components:

```
void TTCN_Runtime::setverdict(verdicttype);
void TTCN_Runtime::setverdict(const VERDICTTYPE&);
verdicttype TTCN_Runtime::getverdict();
```

These functions are the C++ equivalents of TTCN–3 setverdict and getverdict operations. Use them only if your Test Port or C++ function encounters a low-level failure, but it can continue its normal operation (that is, error recovery is not necessary).

Other operators (global functions):

```
boolean operator==(verdicttype par_value,
                   const VERDICTTYPE& other_value); // Equal
boolean operator!=(verdicttype par_value,
                   const VERDICTTYPE& other_value); // Not equal
```

6.3.5. Bitstring

The equivalent C++ class of TTCN–3 type `bitstring` is called `BITSTRING`. The bits of the bit string are stored in an array of unsigned characters. In order to reduce the wasted memory space the bits are packed together, so each character contains eight bits. The first character contains the first eight bits of the bit string; the second byte contains the bits from the 9th up to the 16th, and so on. The first bit of the bit string is the LSB of the first character; the second bit is the second least significant bit of the first character, and so on. The character array is not terminated with a `NUL` character and if the length of the bit string is not a multiple of eight, the unused bits of the last character can contain any value. So the length of the bit string must be always given.

The class `BITSTRING` has the following public member functions:

Table 12. Public member functions of the class `BITSTRING`

Member functions		**Notes**
Constructors	`BITSTRING()`	Initializes to unbound value.
	`BITSTRING(int n_bits, unsigned char *bits_ptr)`	Initializes from a given length and pointer to character array.
	`BITSTRING(const char* str)`	Initializes from a null-terminated character array or string literal
	`BITSTRING(const BITSTRING&)`	Copy constructor.
	`BITSTRING(const BITSTRING_ELEMENT&)`	Initializes from a single bitstring element.
Destructor	`~BITSTRING()`	
Assignment operators	`BITSTRING& operator=(const BITSTRING&)`	Assigns the given value and sets the bound flag.
	`BITSTRING& operator=(const BITSTRING_ELEMENT&)`	Assigns the given single bitstring element.
Comparison operators	`boolean operator==(const BITSTRING&) const`	Returns TRUE if equals
	`boolean operator==(const BITSTRING_ELEMENT&) const`	and FALSE otherwise.
	`boolean operator!=(const BITSTRING&) const`	
	`boolean operator!=(const BITSTRING_ELEMENT&) const`	

Concatenation operator	`BITSTRING operator+(const BITSTRING&) const`	Concatenates two bitstrings.
	`BITSTRING operator+(const BITSTRING_ELEMENT&) const`	Concatenates a bitstring and a bitstring element.
Index operator	`BITSTRING_ELEMENT operator[](int)`	Gives access to the given element. Indexing begins from zero. Index overflow causes dynamic test case error.
	`BITSTRING_ELEMENT operator[](const INTEGER&)`	
	`const BITSTRING_ELEMENT operator[](int) const`	Gives read-only access to the given element.
	`const BITSTRING_ELEMENT operator[](const INTEGER&) const`	
Bitwise operators	`BITSTRING operator~() const`	C++ equivalent of operator not4b. (bitwise negation)
	`BITSTRING operator&(const BITSTRING&) const`	C++ equivalent of operator and4b. (bitwise and)
	`BITSTRING operator&(const BITSTRING_ELEMENT&) const`	
	`BITSTRING operator|(const BITSTRING&) const`	C++ equivalent of operator or4b. (bitwise or)
	`BITSTRING operator|(const BITSTRING_ELEMENT&) const`	
	`BITSTRING operator^(const BITSTRING&) const`	C++ equivalent of operator xor4b. (bitwise xor)
	`BITSTRING operator^(const BITSTRING_ELEMENT&) const`	
Shifting and rotating operators	`BITSTRING operator<<(int) const`	C++ equivalent of operator
	`BITSTRING operator<<(const INTEGER&) const`	<<.(shift left)
	`BITSTRING operator>>(int) const`	C++ equivalent of operator
	`BITSTRING operator>>(const INTEGER&) const`	>>. (shift right)
	`BITSTRING operator<@(int) const`	C++ equivalent of operator
	`BITSTRING operator<@(const INTEGER&) const`	< @. (rotate left)
	`BITSTRING operator>>=(int) const`	C++ equivalent of operator
	`BITSTRING operator>>=(const INTEGER&) const`	@ >. (rotate right)

Casting operator	`operator const unsigned char*() const`	Returns a pointer to the character array.
Other member functions	`int lengthof() const`	Returns the length measured in bits.
	`void log() const`	Puts the value into log. Example: '100011'B.
	`boolean is_bound() const`	Deletes the value, setting it to unbound
	`void clean_up()`	

Using the value of an unbound `BITSTRING` variable for anything will cause dynamic test case error.

`Bitstring element`

The C++ class `BITSTRING_ELEMENT` is the equivalent of the TTCN-3 `bitstring`'s element type (the result of indexing a `bitstring` value). The class does not store the actual bit, only a reference to the original `BITSTRING` object, an index value and a bound flag.

NOTE changing the value of the `BITSTRING_ELEMENT` (through the assignment operator) changes the referenced bit in the original `bitstring` object.

The class `BITSTRING_ELEMENT` has the following public member functions:

Table 13. Public member functions of the class `BITSTRING_ELEMENT`

Member functions		**Notes**
Constructor	`BITSTRING_ELEMENT(boolean par_bound_flag, BITSTRING& par_str_val, int par_bit_pos)`	Initializes the object with an unbound value or a reference to a bit in an existring `BITSTRING` object.
Assignment operators	`BITSTRING_ELEMENT& operator=(const BITSTRING&)`	Sets the referenced bit to the given bitstring of length 1.
	`BITSTRING_ELEMENT& operator=(const BITSTRING_ELEMENT&)`	Sets the referenced bit to the given bitstring element.
Comparison operators	`boolean operator==(const BITSTRING&) const`	Comparison with a bitstring or a bitstring element (the value of the referenced bits is compared, not the references and indexes).
	`boolean operator==(const BITSTRING_ELEMENT&) const`	
	`boolean operator!=(const BITSTRING&) const`	
	`boolean operator!=(const BITSTRING_ELEMENT&) const`	

Concatenation operator	`BITSTRING operator+(const BITSTRING&) const`	Concatenates a bitstring element with a bitstring, or two bitstring elements.
	`BITSTRING operator+(const BITSTRING_ELEMENT&) const`	
Bitwise operators	`BITSTRING operator~() const`	C++ equivalent of operator not4b. (bitwise negation)
	`BITSTRING operator&(const BITSTRING&) const`	C++ equivalent of operator and4b. (bitwise and)
	`BITSTRING operator&(const BITSTRING_ELEMENT&) const`	
	`BITSTRING operator|(const BITSTRING&) const`	C++ equivalent of operator or4b. (bitwise or)
	`BITSTRING operator|(const BITSTRING_ELEMENT&) const`	
	`BITSTRING operator^(const BITSTRING&) const`	C++ equivalent of operator xor4b. (bitwise xor)
	`BITSTRING operator^(const BITSTRING_ELEMENT&) const`	
Shifting and rotating operators	`BITSTRING operator<<(int) const`	C++ equivalent of operator
	`BITSTRING operator<<(const INTEGER&) const`	<<.(shift left)
	`BITSTRING operator>>(int) const`	C++ equivalent of operator
	`BITSTRING operator>>(const INTEGER&) const`	>>. (shift right)
	`BITSTRING operator<⇐(int) const`	C++ equivalent of operator
	`BITSTRING operator<⇐(const INTEGER&) const`	< @. (rotate left)
	`BITSTRING operator>>=(int) const`	C++ equivalent of operator
	`BITSTRING operator>>=(const INTEGER&) const`	@ >. (rotate right)
Other member functions	`boolean get_bit() const`	Returns the referenced bit.
	`void log() const`	Puts the value into log. Example: '1'B.
	`boolean is_bound() const`	Returns whether the value is bound.

Using the value of an unbound `BITSTRING_ELEMENT` variable for anything will cause dynamic test case error.

6.3.6. Hexstring

The equivalent C++ class of TTCN–3 type `hexstring` is called `HEXSTRING`. The hexadecimal digits (nibbles) are stored in an array of unsigned characters. In order to reduce the wasted memory space two nibbles are packed into one character. The first character contains the first two nibbles of the `hexstring`, the second byte contains the third and fourth nibbles, and so on. The hexadecimal digits at odd (first, third, fifth, etc.) positions occupy the lower 4 bits in the characters; the even ones use the upper 4 bits. The character array is never terminated with a `NUL` character, so the length must be always given with the pointer. If the `hexstring` has odd length the unused upper 4 bits of the last character may contain any value.

The class `HEXSTRING` has the following public member functions:

Table 14. Public member functions of the class `HEXSTRING`

Member functions		Notes
Constructors	`HEXSTRING()`	Initializes to unbound value.
	`HEXSTRING(int n_nibbles, const unsigned char *nibbles_ptr)`	Initializes from a given length and pointer to the character array.
	`HEXSTRING(const HEXSTRING&)`	
	`HEXSTRING(const HEXSTRING_ELEMENT&)`	
Destructor	`~HEXSTRING()`	
Assignment operators	`HEXSTRING& operator=(const HEXSTRING&)`	Assigns the given value
	`HEXSTRING& operator=(const HEXSTRING_ELEMENT&)`	
Comparison operators	`boolean operator==(const HEXSTRING&) const`	Returns TRUE if equals and FALSE otherwise.
	`boolean operator==(const HEXSTRING_ELEMENT&) const`	
	`boolean operator!=(const HEXSTRING&) const`	
	`boolean operator!=(const HEXSTRING_ELEMENT&) const`	
Concatenation operator	`HEXSTRING operator+(const HEXSTRING&) const`	Concatenates two hexstrings.
	`HEXSTRING operator+(const HEXSTRING_ELEMENT&) const`	Concatenates a hexstring and a hexstring element.

Member functions		Notes
Index operator	`HEXSTRING_ELEMENT operator[](int)`	Gives access to the given element. Indexing begins from zero. Index overflow causes dynamic test case error.
	`HEXSTRING_ELEMENT operator[](const INTEGER&)`	
	`const HEXSTRING_ELEMENT operator[](int) const`	
	`const HEXSTRING_ELEMENT operator[](const INTEGER&) const`	
Bitwise operators	`HEXSTRING operator~() const`	C++ equivalent of operator not4b. (bitwise negation)
	`HEXSTRING operator&(const HEXSTRING&) const`	C++ equivalent of operator and4b. (bitwise and)
	`HEXSTRING operator&(const HEXSTRING_ELEMENT&) const`	
	`HEXSTRING operator|(const HEXSTRING&) const`	C++ equivalent of operator or4b. (bitwise or)
	`HEXSTRING operator|(const HEXSTRING_ELEMENT&) const`	
	`HEXSTRING operator^(const HEXSTRING&) const`	C++ equivalent of operator xor4b. (bitwise xor)
	`HEXSTRING operator^(const HEXSTRING_ELEMENT&) const`	
Shifting and rotating operators	`HEXSTRING operator<<(int) const`	C++ equivalent of operator
	`HEXSTRING operator<<(const INTEGER&) const`	<<.(shift left)
	`HEXSTRING operator>>(int) const`	C++ equivalent of operator
	`HEXSTRING operator>>(const INTEGER&) const`	>>. (shift right)
	`HEXSTRING operator<⇐(int) const`	C++ equivalent of operator
	`HEXSTRING operator<⇐(const INTEGER&) const`	< @. (rotate left)
	`HEXSTRING operator>>=(int) const`	C++ equivalent of operator
	`HEXSTRING operator>>=(const INTEGER&) const`	@ >. (rotate right)
Casting operator	`operator const unsigned char*() const`	Returns a pointer to the character array. The pointer might be NULL if the length is 0.

Member functions		Notes
Other member functions	`int lengthof() const`	Returns the length measured in nibbles.
	`void log() const`	Puts the value into log. Example: '5A7'H.
	`boolean is_bound() const`	Returns whether the value is bound.
	`void clean_up()`	Deletes the value, setting it to unbound.

Using the value of an unbound `HEXSTRING` variable for anything will cause a dynamic test case error.

`Hexstring element`

The C++ class `HEXSTRING_ELEMENT` is the equivalent of the TTCN-3 `hexstring`'s element type (the result of indexing a `hexstring` value). The class does not store the actual hexadecimal digit (nibble), only a reference to the original HEXSTRING object, an index value and a bound flag.

NOTE changing the value of the `HEXSTRING_ELEMENT` (through the assignment operator) changes the referenced nibble in the original `hexstring` object.

The class `HEXSTRING_ELEMENT` has the following public member functions:

Table 15. Public member functions of the class `HEXSTRING_ELEMENT`

Member functions		Notes
Constructor	`HEXSTRING_ELEMENT(boolean par_bound_flag, HEXSTRING& par_str_val, int par_nibble_pos)`	Initializes the object with an unbound value or a reference to a nibble in an existring HEXSTRING object.
Assignment operators	`HEXSTRING_ELEMENT& operator=(const HEXSTRING&)`	Sets the referenced nibble to the given hexstring of length 1.
	`HEXSTRING_ELEMENT& operator=(const HEXSTRING_ELEMENT&)`	Sets the referenced nibble to the given hexstring element.
Comparison operators	`boolean operator==(const HEXSTRING&) const`	Comparison with a hexstring or a hexstring element (the value of the referenced nibbles is compared, not the references and indexes).
	`boolean operator==(const HEXSTRING_ELEMENT&) const`	
	`boolean operator!=(const HEXSTRING&) const`	
	`boolean operator!=(const HEXSTRING_ELEMENT&) const`	

Concatenation operator	`HEXSTRING operator+(const HEXSTRING&) const`	Concatenates a hexstring element with a hexstring, or two hexstring elements.
	`HEXSTRING operator+(const HEXSTRING_ELEMENT&) const`	
Bitwise operators	`HEXSTRING operator~() const`	C++ equivalent of operator not4b. (bitwise negation)
	`HEXSTRING operator&(const HEXSTRING&) const`	C++ equivalent of operator and4b. (bitwise and)
	`HEXSTRING operator&(const HEXSTRING_ELEMENT&) const`	
	`HEXSTRING operator|(const HEXSTRING&) const`	C++ equivalent of operator or4b. (bitwise or)
	`HEXSTRING operator|(const HEXSTRING_ELEMENT&) const`	
	`HEXSTRING operator^(const HEXSTRING&) const`	C++ equivalent of operator xor4b. (bitwise xor)
	`HEXSTRING operator^(const HEXSTRING_ELEMENT&) const`	
Shifting and rotating operators	`HEXSTRING operator<<(int) const`	C++ equivalent of operator
	`HEXSTRING operator<<(const INTEGER&) const`	<<.(shift left)
	`HEXSTRING operator>>(int) const`	C++ equivalent of operator
	`HEXSTRING operator>>(const INTEGER&) const`	>>. (shift right)
	`HEXSTRING operator<⇐(int) const`	C++ equivalent of operator
	`HEXSTRING operator<⇐(const INTEGER&) const`	< @. (rotate left)
	`HEXSTRING operator>>=(int) const`	C++ equivalent of operator
	`HEXSTRING operator>>=(const INTEGER&) const`	@ >. (rotate right)
Other member functions	`unsigned char get_nibble() const`	Returns the referenced nibble (stored in the lower 4 bits of the returned character).
	`void log() const`	Puts the value into log. Example: '8'H.
	`boolean is_bound() const`	Returns whether the value is bound.

Using the value of an unbound `HEXSTRING_ELEMENT` variable for anything will cause dynamic test case error.

6.3.7. Octetstring

The equivalent C++ class of TTCN–3 type octetstring is called OCTETSTRING. The octets are stored in an array of unsigned characters. Each character contains one octet; the first character is the first octet of the string. The character array is not terminated by a NUL character, so the length of the octet string must be always given.

The class OCTETSTRING has the following public member functions:

Table 16. Public member functions of the class OCTETSTRING

Member functions		Notes
Constructors	OCTETSTRING()	Initializes to unbound value.
	OCTETSTRING(int n_octets, const unsigned char *octets_ptr)	Initializes from a given length and pointer to character array.
	OCTETSTRING(const char* str)	Initializes from a null-terminated character array or string literal
	OCTETSTRING(const OCTETSTRING&)	Copy constructor.
	OCTETSTRING(const OCTETSTRING_ELEMENT&)	Initializes from a single octetstring element.
Destructor	~OCTETSTRING()	
Assignment operators	OCTETSTRING& operator=(const OCTETSTRING&)	Assigns the given value and sets the bound flag.
	OCTETSTRING& operator=(const OCTETSTRING_ELEMENT&)	Assigns the given octetstring element.
Comparison operators	boolean operator==(const OCTETSTRING&) const	Returns TRUE if equals
	boolean operator==(const OCTETSTRING_ELEMENT&) const	and FALSE otherwise.
	boolean operator!=(const OCTETSTRING&) const	
	boolean operator!=(const OCTETSTRING_ELEMENT&) const	
Concatenation operator	OCTETSTRING operator+(const OCTETSTRING&) const	Concatenates two octetstrings.
	OCTETSTRING operator+(const OCTETSTRING_ELEMENT&) const	Concatenates an octetstring and an octetstring element.
	OCTETSTRING& operator+=(const OCTETSTRING&) const	Appends an octetstring to this one.
	OCTETSTRING& operator+=(const OCTETSTRING_ELEMENT&) const	Appends an octetstring element to this octetstring.

Member functions		Notes
Index operator	`OCTETSTRING_ELEMENT operator[](int)`	Gives access to the given element. Indexing begins from zero. Index overflow causes dynamic test case error.
	`OCTETSTRING_ELEMENT operator[](const INTEGER&)`	
	`const OCTETSTRING_ELEMENT operator[](int) const`	Gives read-only access to the given element.
	`const OCTETSTRING_ELEMENT operator[](const INTEGER&) const`	
Bitwise operators	`OCTETSTRING operator~() const`	C++ equivalent of operator not4b.(bitwise negation)
	`OCTETSTRING operator&(const OCTETSTRING&) const`	C++ equivalent of operator and4b. (bitwise and)
	`OCTETSTRING operator&(const OCTETSTRING_ELEMENT&) const`	
	`OCTETSTRING operator|(const OCTETSTRING&) const`	C++ equivalent of operator or4b. (bitwise or)
	`OCTETSTRING operator|(const OCTETSTRING_ELEMENT&) const`	
	`OCTETSTRING operator^(const OCTETSTRING&) const`	C++ equivalent of operator xor4b. (bitwise xor)
	`OCTETSTRING operator^(const OCTETSTRING_ELEMENT&) const`	
Shifting and rotating operators	`OCTETSTRING operator<<(int) const`	C++ equivalent of operator <<.
	`OCTETSTRING operator<<(const INTEGER&) const`	(shift left)
	`OCTETSTRING operator>>(int) const`	C++ equivalent of operator >>.
	`OCTETSTRING operator>>(const INTEGER&) const`	(shift right)
	`OCTETSTRING operator<@(int) const`	C++ equivalent of operator < @.
	`OCTETSTRING operator<@(const INTEGER&) const`	(rotate left)
	`OCTETSTRING operator>>@(int) const`	C++ equivalent of operator @ >.
	`OCTETSTRING operator>>@(const INTEGER&) const`	(rotate right)

Member functions		Notes
Casting operator	`operator const unsigned char*() const`	Returns a pointer to the character array. The pointer might be NULL if the length is 0.
Other member functions	`int lengthof() const`	Returns the length measured in octets.
	`void log() const`	Puts the value into log. Like '073CF0'O.
	`boolean is_bound() const`	Returns whether the value is bound.
	`void clean_up()`	Deletes the value, setting it to unbound.

Using the value of an unbound `OCTETSTRING` variable for anything will cause dynamic test case error.

`Octetstring element`

The C++ class `OCTETSTRING_ELEMENT` is the equivalent of the TTCN-3 `octetstring`'s element type (the result of indexing an `octetstring` value). The class does not store the actual octet, only a reference to the original OCTETSTRING object, an index value and a bound flag.

NOTE changing the value of the OCTETSTRING_ELEMENT (through the assignment operator) changes the referenced octet in the original `octetstring` object.

The class `OCTETSTRING_ELEMENT` has the following public member functions:

Table 17. Public member functions of the class `OCTETSTRING_ELEMENT`

Member functions		Notes
Constructor	`OCTETSTRING_ELEMENT(boolean par_bound_flag, OCTETSTRING& par_str_val, int par_octet_pos)`	Initializes the object with an unbound value or a reference to an octet in an existring OCTETSTRING object.
Assignment operators	`OCTETSTRING_ELEMENT& operator=(const OCTETSTRING&)`	Sets the referenced octet to the given octetstring of length 1.
	`OCTETSTRING_ELEMENT& operator=(const OCTETSTRING_ELEMENT&)`	Sets the referenced octet to the given octetstring element.

Member functions		Notes
Comparison operators	`boolean operator==(const OCTETSTRING&) const`	Comparison with an octetstring or an octetstring element (the value of the referenced octets is compared, not the references and indexes).
	`boolean operator==(const OCTETSTRING_ELEMENT&) const`	
	`boolean operator!=(const OCTETSTRING&) const`	
	`boolean operator!=(const OCTETSTRING_ELEMENT&) const`	
Concatenation operator	`OCTETSTRING operator+(const OCTETSTRING&) const`	Concatenates an octetstring element with an octetstring, or two octetstring elements.
	`OCTETSTRING operator+(const OCTETSTRING_ELEMENT&) const`	
Bitwise operators	`OCTETSTRING operator~() const`	C++ equivalent of operator (bitwise negation)
	`OCTETSTRING operator&(const OCTETSTRING&) const`	C++ equivalent of operator and4b. (bitwise and)
	`OCTETSTRING operator&(const OCTETSTRING_ELEMENT&) const`	
	`OCTETSTRING operator|(const OCTETSTRING&) const`	C++ equivalent of operator or4b. (bitwise or)
	`OCTETSTRING operator|(const OCTETSTRING_ELEMENT&) const`	
	`OCTETSTRING operator^(const OCTETSTRING&) const`	C++ equivalent of operator xor4b. (bitwise xor)
	`OCTETSTRING operator^(const OCTETSTRING_ELEMENT&) const`	

Member functions		Notes
Shifting and rotating operators	`OCTETSTRING operator<<(int) const`	C++ equivalent of operator <<.
	`OCTETSTRING operator<<(const INTEGER&) const`	(shift left)
	`OCTETSTRING operator>>(int) const`	C++ equivalent of operator >>.
	`OCTETSTRING operator>>(const INTEGER&) const`	(shift right)
	`OCTETSTRING operator<@(int) const`	C++ equivalent of operator < @.
	`OCTETSTRING operator<@(const INTEGER&) const`	(rotate left)
	`OCTETSTRING operator>>@(int) const`	C++ equivalent of operator @ >.
	`OCTETSTRING operator>>@(const INTEGER&) const`	(rotate right)
Other member functions	`unsigned char get_octet() const`	Returns the referenced octet.
	`void log() const`	Puts the value into log. Example: '3C'O.
	`boolean is_bound() const`	Returns whether the value is bound.

Using the value of an unbound `OCTETSTRING_ELEMENT` variable for anything will cause dynamic test case error.

6.3.8. `Char`

The `char` type, which has been removed from the TTCN–3 standard, is no longer supported by the run-time environment. The compiler substitutes all occurrences of `char` type with type `charstring` automatically.

To provide partial backward compatibility for older Test Ports that might have used the type `char`, `CHAR` is a typedef alias to class `CHARSTRING` in C++.

6.3.9. `Charstring`

The equivalent C++ class of TTCN–3 type `charstring` is called `CHARSTRING`. The characters are stored in a `NUL` character terminated array; thus, giving the length in the constructor and other operations is optional.

The class `CHARSTRING` has the following public member functions:

Table 18. Public member functions of the class `CHARSTRING`

Member functions	Notes

Constructors	`CHARSTRING()`	Initializes to unbound value.
	`CHARSTRING(char)`	Initializes from a single character.
	`CHARSTRING(int n_chars, const char *chars_ptr)`	Initializes from a given length and pointer to character array.
	`CHARSTRING(const char *chars_ptr)`	Initializes from a given character array. The end is noted by a NUL character.
	`CHARSTRING(const CHARSTRING&)`	Copy constructor.
	`CHARSTRING(const CHARSTRING_ELEMENT&)`	Initializes from a charstring element.
Destructor	`~CHARSTRING()`	
Assignment operators	`CHARSTRING& operator=(const CHARSTRING&)`	Assigns the given value and sets the bound flag.
	`CHARSTRING& operator=(const char *)`	Assigns the NUL terminated string.
	`CHARSTRING& operator=(const CHARSTRING_ELEMENT&)`	Assigns the given charstring element.
	`CHARSTRING& operator=(const UNIVERSAL_CHARSTRING&)`	Assigns the given universal charstring value.
Comparison operators	`boolean operator==(const CHARSTRING&) const`	Returns TRUE if equals and FALSE otherwise.
	`boolean operator==(const char *) const`	Compares to the NUL terminated string.
	`boolean operator==(const CHARSTRING_ELEMENT&) const`	Comparison with a charstring element.
	`boolean operator==(const UNIVERSAL_CHARSTRING&) const`	Comparison with a universal charstring.
	vboolean operator==(const UNIVERSAL_CHARSTRING_ELEMENT&) const`	Comparison with a universal charstring element.
	`boolean operator!=(const CHARSTRING&) const`	
	`boolean operator!=(const char *) const`	
	`boolean operator!=(const CHARSTRING_ELEMENT&) const`	

Concatenation operator	`CHARSTRING operator+(const CHARSTRING&) const`	Concatenates two charstrings.
	`CHARSTRING operator+(const char *) const`	Concatenates with a NUL terminated string.
	`CHARSTRING operator+(const CHARSTRING_ELEMENT) const`	Concatenates with a charstring element.
	`UNIVERSAL_CHARSTRING operator+(const UNIVERSAL_CHARSTRING&) const`	Concatenates with a universal charstring.
	`UNIVERSAL_CHARSTRING operator+(const UNIVERSAL_CHARSTRING_ELEMENT&) const`	Concatenates with a universal charstring element.
	`CHARSTRING operator+=(char)`	Appends a character.
	`CHARSTRING operator+=(const char *)`	Appends a NUL terminated string.
	`CHARSTRING operator+=(const CHARSTRING&)`	Appends a charstring.
	`CHARSTRING operator+=(const CHARSTRING_ELEMENT&)`	Appends a charstring element.
Index operator	`CHARSTRING_ELEMENT operator[](int)`	Gives access to the given element. Indexing begins from zero. Index overflow causes dynamic test case error.
	`CHARSTRING_ELEMENT operator[](const INTEGER&)`	
	`const CHARSTRING_ELEMENT operator[](int) const`	Gives read-only access to the given element.
	`const CHARSTRING_ELEMENT operator[](const INTEGER&) const`	
Rotating operators	`CHARSTRING operator<@(int) const`	C++ equivalent of operator < @.(rotate left)
	`CHARSTRING operator<@(const INTEGER&) const`	
	`CHARSTRING operator>>@(int) const`	C++ equivalent of operator @ >. (rotate right)
	`CHARSTRING operator>>@(const INTEGER&) const`	
Casting operator	`operator const char*() const`	Returns a pointer to the character array. The string is always terminated by NUL.

Other member functions	`int lengthof() const`	Returns the length measured in characters not including the terminator NUL.
	`void log() const`	Puts the value into log. Example: ”abc”.
	`boolean is_bound() const`	Returns whether the value is bound.

The comparison, concatenation and rotating operators are also available as global functions for that case when the left side is `const char*` and the right side is `CHARSTRING`.

The log() member function uses single character output for regular characters, but special characters (such as the quotation mark, backslash or newline characters) are printed using the escape sequences of the C language. Non-printable control characters are printed in TTCN–3 quadruple notation, where the first three octets are always zero. The concatenation operator (`&`) is used between the fragments when necessary. Note that the output does not always conform to TTCN–3 Core Language syntax, but it is always recognized by both our compiler and the configuration file parser.

Using the value of an unbound `CHARSTRING` variable for anything will cause dynamic test case error.

Other operators (global functions):

```
boolean operator==(const char* string_value,
                   const CHARSTRING& other_value);           // Equal
boolean operator==(const char* string_value,
                   const CHARSTRING_ELEMENT& other_value);   // Equal
boolean operator!=(const char* string_value,
                   const CHARSTRING& other_value);           // Not equal
boolean operator!=(const char* string_value,
                   const CHARSTRING_ELEMENT& other_value);   // Not equal
CHARSTRING operator+(const char* string_value,
                     const CHARSTRING& other_value);         // Concatenation
CHARSTRING operator+(const char* string_value,
                     const CHARSTRING_ELEMENT& other_value); // Concatenation
```

`Charstring element`

The C++ class `CHARSTRING_ELEMENT` is the equivalent of the TTCN-3 `charstring`’s element type (the result of indexing a `charstring` value). The class does not store the actual character, only a reference to the original CHARSTRING object, an index value and a bound flag.

NOTE changing the value of the `CHARSTRING_ELEMENT` (through the assignment operator) changes the referenced character in the original `charstring` object.

The class `CHARSTRING_ELEMENT` has the following public member functions:

Table 19. Public member functions of the class `CHARSTRING_ELEMENT`

Member functions		Notes
Constructor	`CHARSTRING_ELEMENT(boolean par_bound_flag, CHARSTRING& par_str_val, int par_char_pos)`	Initializes the object with an unbound value or a reference to a character in an existring CHARSTRING object.
Assignment operators	`CHARSTRING_ELEMENT& operator=(const char*)`	Sets the referenced character to the given null-terminated string of length 1.
	`CHARSTRING_ELEMENT& operator=(const CHARSTRING&)`	Sets the referenced character to the given charstring of length 1.
	`CHARSTRING_ELEMENT& operator=(const CHARSTRING_ELEMENT&)`	Sets the referenced character to the given charstring element.
Comparison operators	`boolean operator==(const char*) const`	Comparison with a null-terminated string, a charstring, a universal charstring, a charstring element or a universal charstring element (when comparing element types, the value of the referenced characters is compared, not the references and indexes).
	`boolean operator==(const CHARSTRING&) const`	
	`boolean operator==(const CHARSTRING_ELEMENT&) const`	
	`boolean operator==(const UNIVERSAL_CHARSTRING&) const`	
	`boolean operator==(const UNIVERSAL_CHARSTRING_ELEMENT&) const`	
	`boolean operator!=(const char*) const`	
	`boolean operator!=(const CHARSTRING&) const`	
	`boolean operator!=(const CHARSTRING_ELEMENT&) const`	

Concatenation operator	`CHARSTRING operator+(const char*) const`	Concatenates this object with a null-terminated string, a charstring, a charstring element, a universal charstring or a universal charstring element.
	`CHARSTRING operator+(const CHARSTRING&) const`	
	`CHARSTRING operator+(const CHARSTRING_ELEMENT&) const`	
	`UNIVERSAL_CHARSTRING operator+(const UNIVERSAL_CHARSTRING&) const`	
	`UNIVERSAL_CHARSTRING operator+(const UNIVERSAL_CHARSTRING_ELEMENT&) const`	
Rotating operators	`CHARSTRING operator<<=(int) const`	C++ equivalent of operator << @.(rotate left)
	`CHARSTRING operator<<=(const INTEGER&) const`	
	`CHARSTRING operator>>=(int) const`	C++ equivalent of operator @ >>. (rotate right)
	`CHARSTRING operator>>=(const INTEGER&) const`	
Other member functions	`char get_char() const`	Returns the referenced character.
	`void log() const`	Puts the value into log. Example: “a”.
	`boolean is_bound() const`	Returns whether the value is bound.

Using the value of an unbound `CHARSTRING_ELEMENT` variable for anything will cause dynamic test case error.

6.3.10. `Universal char`

This obsolete TTCN–3 type is converted automatically to `universal charstring` in the parser.

6.3.11. `Universal charstring`

Each character of a `universal charstring` value is represented in the following C structure defined in the Base Library:

```
struct universal_char {
  unsigned char uc_group, uc_plane, uc_row, uc_cell;
};
```

The four components of the quadruple (that is, group, plane, row and cell) are stored in fields `uc_group`, `uc_plane`, `uc_row` and `uc_cell`, respectively. All fields are 8bit unsigned numeric values with the possible value range 0 .. 255.

In case of single-octet characters, which can be also given in TTCN–3 charstring notation (between quotation marks), the fields `uc_group`, `uc_plane`, `uc_row` are set to zero. If tuple notation was used for an ASN.1 string value fields `uc_row` and `uc_cell` carry the tuple and the others are set to zero.

Except when performing encoding or decoding, the run-time environment does not check whether the quadruples used in the following API represent valid character positions according to [8]. Moreover, if ASN.1 multi-octet character string values are used, it is not verified whether the elements of such strings are permitted characters of the corresponding string type.

The C++ equivalent of TTCN–3 type `universal charstring` is implemented in class `UNIVERSAL_CHARSTRING`. The characters of the string are stored in an array of structure `universal_char`. The array returned by the casting operator is not terminated with a special character, thus, the length of the string must be always considered when doing operations with the array. The length of the string, which can be obtained by using member function `lengthof()`, is measured in characters (quadruples) and not bytes.

For the more convenient usage the strings containing only single-octet characters can also be used with class `UNIVERSAL_CHARSTRING`. Therefore some polymorphic member functions and operators have variants that take `const char*` as argument. In these member functions the characters of the `NUL` character terminated string are implicitly converted to quadruples with group, plane and row fields set to zero. `NULL` pointer as argument means the empty string for these functions.

The class `UNIVERSAL_CHARSTRING` has the following public member functions:

Table 20. Public member functions of the class `UNIVERSAL_CHARSTRING`

Member functions	Notes

Constructors	`UNIVERSAL_CHARSTRING()`	Initializes to unbound value.
	`UNIVERSAL_CHARSTRING (unsigned char group, unsigned char plane, unsigned char row, unsigned char cell)`	Constructs a string containing one character formed from the given quadruple.
	`UNIVERSAL_CHARSTRING (const universal_char&)`	Constructs a string containing the given single character.
	`UNIVERSAL_CHARSTRING (int n_uchars, const universal_char *uchars_ptr)`	Constructs a string from an array by taking the given number of single-octet characters.
	`UNIVERSAL_CHARSTRING (const char *chars_ptr)`	Constructs a string from a NUL terminated array of single-octet characters.
	`UNIVERSAL_CHARSTRING (int n_chars, const char *chars_ptr)`	Constructs a string from a given number of single-octet characters.
	`UNIVERSAL_CHARSTRING (const CHARSTRING&)`	Constructs a universal charstring from a charstring value.
	`UNIVERSAL_CHARSTRING (const CHARSTRING_ELEMENT&)`	Constructs a string containing the given singe charstring element.
	`UNIVERSAL_CHARSTRING (const UNIVERSAL_CHARSTRING&)`	Copy constructor.
	`UNIVERSAL_CHARSTRING (const UNIVERSAL_CHARSTRING_ELEMENT&)`	Constructs a string containing the given singe universal charstring element.
Destructor	`~UNIVERSAL_CHARSTRING()`	
Assignment operators	`UNIVERSAL_CHARSTRING& operator= (const UNIVERSAL_CHARSTRING&)`	Assigns another string.
	`UNIVERSAL_CHARSTRING& operator= (const universal_char&)`	Assigns a single character.
	`UNIVERSAL_CHARSTRING& operator= (const char*)`	Assigns a NUL terminated single-octet string.
	`UNIVERSAL_CHARSTRING& operator= (const CHARSTRING&)`	Assigns a charstring.
	`UNIVERSAL_CHARSTRING& operator= (const CHARSTRING_ELEMENT&)`	Assigns a single charstring element.
	`UNIVERSAL_CHARSTRING& operator= (const UNIVERSAL_CHARSTRING_ELEMENT&)`	Assigns a single universal charstring element.

Comparison operators	`boolean operator==(const UNIVERSAL_CHARSTRING&) const`	Returns TRUE if the strings are identical or FALSE otherwise.
	`boolean operator==(const universal_char&) const`	Compares to a single character.
	`boolean operator==(const char*) const`	Compares to a NUL terminated printable string.
	`boolean operator==(const CHARSTRING&) const`	Compares to a charstring.
	`boolean operator==(const CHARSTRING_ELEMENT&) const`	Compares to a charstring element.
	`boolean operator==(const UNIVERSAL_CHARSTRING_ELEMENT&) const`	Compares to a universal charstring element.
	`boolean operator!=(const UNIVERSAL_CHARSTRING&) const`	
	`boolean operator!= (const universal_char&) const`	
	`boolean operator!=(const char*) const`	
	`boolean operator!=(const CHARSTRING&)`	
	`boolean operator!=(const CHARSTRING_ELEMENT&) const`	
	`boolean operator!=(const UNIVERSAL_CHARSTRING_ELEMENT&) const`	
Concatenation operator	`UNIVERSAL_CHARSTRING operator+(const UNIVERSAL_CHARSTRING&) const`	Concatenates two strings.
	`UNIVERSAL_CHARSTRING operator+(const universal_char&) const`	Concatenates a single character.
	`UNIVERSAL_CHARSTRING operator+(const char*) const`	Concatenates a NUL terminated single-octet string.
	`UNIVERSAL_CHARSTRING operator+(const CHARSTRING&) const`	Concatenates a charstring.
	`UNIVERSAL_CHARSTRING operator+(const CHARSTRING_ELEMENT&) const`	Concatenates a charstring element.
	`UNIVERSAL_CHARSTRING operator+(const UNIVERSAL_CHARSTRING_ELEMENT&) const`	Concatenates a universal charstring element.

Index operator	`UNIVERSAL_CHARSTRING_ELEMENT operator[](int)`	Gives access to the given element. Indexing begins from zero. Index overflow causes dynamic test case error.
	`UNIVERSAL_CHARSTRING_ELEMENT operator[](const INTEGER&)`	
	`const UNIVERSAL_CHARSTRING_ELEMENT operator[](int) const`	Gives read-only access to the given element.
	`const UNIVERSAL_CHARSTRING_ELEMENT operator[](const INTEGER&) const`	
Rotating operators	`UNIVERSAL_CHARSTRING operator<<=(int) const`	C++ equivalent of operator <@(rotate left).
	`UNIVERSAL_CHARSTRING operator<<=(const INTEGER&) const`	
	`UNIVERSAL_CHARSTRING operator>>=(int) const`	C++ equivalent of operator @ >(rotate right).
	`UNIVERSAL_CHARSTRING operator>>=(const INTEGER&) const`	
Casting operator	`operator const universal_char*() const`	Returns a pointer to the array of characters. There is no terminator character at the end.
UTF-8 encoding and decoding	`void encode_utf8(TTCN_Buffer& buf) const`	Appends the UTF-8 representation of the string to the given buffer
	`void decode_utf8(int n_octets, const unsigned char *octets_ptr)`	
Other member functions	`int lengthof() const`	Returns the length measured in characters.
	`boolean is_bound() const `	Returns whether the value is bound.
	`void log() const`	Puts the value into log. See below.
	`void clean_up()`	Deletes the value, setting it to unbound.

The comparison and concatenation operators are also available as global functions for that case when the left operand is a single-octet string (`const char*`) or a single character (`const universal_char&`) and the right side is `UNIVERSAL_CHARSTRING` value. Using the value of an unbound `UNIVERSAL_CHARSTRING` variable for anything causes dynamic test case error.

The `UNIVERSAL_CHARSTRING` variable used with the `decode_utf8()` method must be newly constructed (unbound) or `clean_up()` must have been called, otherwise a memory leak will occur.

The logged printout of universal charstring values is compatible with the TTCN–3 notation for such strings. The format to be used depends on the contents of the string. Each character (quadruple) is classified whether it is directly printable or not. The string is fragmented based on this classification. Each fragment consists of either a single non-printable character or a maximal length contiguous sequence of printable characters. The fragments are logged one after another separated by an `&` character (concatenation operator). The printable fragments use the normal charstring notation; the non-printable characters are logged in the TTCN–3 quadruple notation. An empty universal charstring value is represented by a pair of quotation marks (like in case of empty charstring values).

An example printout in the log can be the following. The string consists of two fragments of printable characters and a non-printable quadruple, which stands for Hungarian letter "ű":

```
"Character " & char(0, 0, 1, 113) & " is a letter of Hungarian alphabet"
```

Other operators (global functions):

```
boolean operator==(const universal_char& left_value,
                   const universal_char& right_value);                 // Equal
boolean operator==(const universal_char& uchar_value,
                   const UNIVERSAL_CHARSTRING& other_value);           // Equal
boolean operator==(const char* string_value,
                   const UNIVERSAL_CHARSTRING& other_value);           // Equal
boolean operator==(const universal_char& uchar_value,
                   const UNIVERSAL_CHARSTRING_ELEMENT& other_value);   // Equal
boolean operator==(const char* string_value,
                   const UNIVERSAL_CHARSTRING_ELEMENT& other_value);   // Equal
boolean operator!=(const universal_char& left_value,
                   const universal_char& right_value);             // Not equal
boolean operator!=(const universal_char& uchar_value,
                   const UNIVERSAL_CHARSTRING& other_value);       // Not equal
boolean operator!=(const char* string_value,
                   const UNIVERSAL_CHARSTRING& other_value);       // Not equal
boolean operator!=(const universal_char& uchar_value,
              const UNIVERSAL_CHARSTRING_ELEMENT& other_value);      // Not equal
boolean operator!=(const char* string_value,
              const UNIVERSAL_CHARSTRING_ELEMENT& other_value);      // Not equal
boolean operator<(const universal_char& left_value,
         const universal_char& right_value& other_value); // Character comparison
UNIVERSAL_CHARSTRING operator+(const universal_char& uchar_value,
             const UNIVERSAL_CHARSTRING& other_value);           // Concatenation
UNIVERSAL_CHARSTRING operator+(const char* string_value,
             const UNIVERSAL_CHARSTRING& other_value);           // Concatenation
UNIVERSAL_CHARSTRING operator+(const universal_char& uchar_value,
             const UNIVERSAL_CHARSTRING_ELEMENT& other_value);   // Concatenation
UNIVERSAL_CHARSTRING operator+(const char* string_value,
             const UNIVERSAL_CHARSTRING_ELEMENT& other_value);   // Concatenation
```

Universal charstring element

The C++ class `UNIVERSAL_CHARSTRING_ELEMENT` is the equivalent of the TTCN-3 `universal charstring`'s element type (the result of indexing a `universal charstring` value). The class does not store the actual character, only a reference to the original `UNIVERSAL_CHARSTRING` object, an index value and a bound flag.

NOTE	changing the value of the `UNIVERSAL_CHARSTRING_ELEMENT` (through the assignment operator) changes the referenced character in the original `universal charstring` object.

The class `UNIVERSAL_CHARSTRING_ELEMENT` has the following public member functions:

Table 21. Public member functions of the class `UNIVERSAL_CHARSTRING_ELEMENT`

Member functions	Notes

Constructor	`UNIVERSAL_CHARSTRING_ELEMENT(boolean par_bound_flag, UNIVERSAL_CHARSTRING& par_str_val, int par_uchar_pos)`	Initializes the object with an unbound value or a reference to a character in an existring UNIVERSAL_CHARSTRING object.
Assignment operators	`UNIVERSAL_CHARSTRING_ELEMENT& operator=(const universal_char&)`	Sets the referenced character to the given universal character.
	`UNIVERSAL_CHARSTRING_ELEMENT& operator=(const char*)`	
	`UNIVERSAL_CHARSTRING_ELEMENT& operator=(const CHARSTRING&)`	
	`UNIVERSAL_CHARSTRING_ELEMENT& operator=(const CHARSTRING_ELEMENT&)`	
	`UNIVERSAL_CHARSTRING_ELEMENT& operator=(const UNIVERSAL_CHARSTRING&)`	
	`UNIVERSAL_CHARSTRING_ELEMENT& operator=(const UNIVERSAL_CHARSTRING_ELEMENT&)`	

Comparison operators	`boolean operator==(const universal_char&) const`	Comparison with a universal character, a null-terminated string, a charstring, a universal charstring, a charstring element or a universal charstring element (when comparing element types, the value of the referenced characters is compared, not the references and indexes).
	`boolean operator==(const char*) const`	
	`boolean operator==(const CHARSTRING&) const`	
	`boolean operator==(const CHARSTRING_ELEMENT&) const`	
	`boolean operator==(const UNIVERSAL_CHARSTRING&) const`	
	`boolean operator==(const UNIVERSAL_CHARSTRING_ELEMENT&) const`	
	`boolean operator!=(const universal_char&) const`	
	`boolean operator!=(const char*) const`	
	`boolean operator!=(const CHARSTRING&) const`	
	`boolean operator!=(const CHARSTRING_ELEMENT&) const`	
	`boolean operator!=(const UNIVERSAL_CHARSTRING&) const`	
	`boolean operator!=(const UNIVERSAL_CHARSTRING_ELEMENT&) const`	

Concatenation operator	`CHARSTRING operator+(const universal_char&) const`	Concatenates this object with a universal character, a null-terminated string, a charstring, a charstring element, a universal charstring or a universal charstring element.
	`CHARSTRING operator+(const char*) const`	
	`CHARSTRING operator+(const CHARSTRING&) const`	
	`CHARSTRING operator+(const CHARSTRING_ELEMENT&) const`	
	`UNIVERSAL_CHARSTRING operator+(const UNIVERSAL_CHARSTRING&) const`	
	`UNIVERSAL_CHARSTRING operator+(const UNIVERSAL_CHARSTRING_ELEMENT&) const`	
Rotating operators	`UNIVERSAL_CHARSTRING operator<<=(int) const`	C++ equivalent of operator << @(rotate left).
	`UNIVERSAL_CHARSTRING operator<<=(const INTEGER&) const`	
	`UNIVERSAL_CHARSTRING operator>>=(int) const`	C++ equivalent of operator @ >> (rotate right).
	`UNIVERSAL_CHARSTRING operator>>=(const INTEGER&) const`	
Other member functions	`const universal_char& get_char() const`	Returns the referenced character.
	`void log() const`	Puts the value into log. Example: “a” or char(0, 0, 1, 113).
	`boolean is_bound() const`	Returns whether the value is bound.

Using the value of an unbound `UNIVERSAL_CHARSTRING_ELEMENT` variable for anything will cause dynamic test case error.

6.3.12. Object Identifier Type

The object identifier type of TTCN–3 (`objid`) is implemented in class OBJID. In the run-time environment the components of object identifier values are represented in NumberForm, that is, in integer values. The values of components are stored in an array with a given length. The type of the components is specified with a `typedef`, `objid_element`. Class `OBJID` has the following member

functions.

Table 22. Public member functions of the class `OBJID`

Member functions		Notes
Constructors	`OBJID()`	Initializes to unbound value.
	`OBJID(int n_components, const objid_element *components_ptr)`	Initializes the number of components to n components and copies all components from an array of integers starting at components_ptr.
	`OBJID(int n_components, ⋯)`	Initializes the number of components to n_components. The components themselves shall be given as additional integer arguments after each other, starting with the first one.
	`OBJID(const OBJID&)`	Copy constructor.
Destructor	`~OBJID()`	
Assignment operator	`OBJID& operator=(const OBJID&)`	Assigns the given value and sets the bound flag.
Comparison operators	`boolean operator==(const OBJID&) const`	Returns TRUE if the two values are equal and FALSE otherwise.
	`boolean operator!=(const OBJID&) const`	
Indexing operators	`objid_element& operator[](int i)`	Returns a reference to the i^{th} component.
	`const objid_element & operator[](int i) const`	Returns a read-only reference to the i^{th} component.
Casting operator	`operator const objid_element *() const`	Returns a pointer to the read-only array of components.
Other member functions	`int lengthof() const`	Returns the number of components.
	`void log() const`	Puts the value into log in NumberForm. Like this: “objid 0 4 0 ”.
	`boolean is_bound() const`	Returns whether the value is bound.
	`void clean_up()`	Deletes the value, setting it to unbound.

NOTE	The constructor with variable number of arguments is useful in situations when the number of components is constant and known at compile time.

Using the value of an unbound `OBJID` variable for anything will cause dynamic test case error.

6.3.13. Component References

TTCN–3 variables the types of which are defined as component types are used for storing component references to PTCs. The internal representation of component references are test tool dependent, our test executor handles them as small integer numbers.

All TTCN–3 component types are mapped to the same C++ class, which is called COMPONENT, using `typedef` aliases. We also use an ancillary C type called `component`, which is defined as an alias for `int`:

```
typedef int component;
```

There are some predefined constants of component references in TTCN–3. These are defined as C preprocessor macros in the following way:

Table 23. Predefined component references

TTCN–3 constant	Preprocessor symbol	Numeric value
null	NULL	COMPREF 0
mtc	MTC	COMPREF 1
system	SYSTEM	COMPREF 2

The class `COMPONENT` has the following public member functions:

Table 24. Public member functions of the class `COMPONENT`

Member functions		Notes
Constructors	`COMPONENT()`	Initializes to unbound value.
	`COMPONENT(component)`	Initializes to a given value.
	`COMPONENT(const COMPONENT&)`	Copy constructor.
Destructor	`COMPONENT()`	
Assignment operators	`COMPONENT& operator=(component)`	Assigns the given value
	`COMPONENT& operator=(const COMPONENT&)`	and sets the bound flag.

Comparison operators	`boolean operator==(component) const`	Returns TRUE if equals
	`boolean operator==(const COMPONENT&) const`	and FALSE otherwise.
	`boolean operator!=(component) const`	
	`boolean operator!=(const COMPONENT&) const`	
Casting operator	`operator component() const`	Returns the value.
Other member functions	`void log() const`	Puts the value into log in decimal form or in symbolic format for special constants. Like 3 or mtc.
	`boolean is_bound() const`	Returns whether the value is bound.
	`void clean_up()`	Deletes the value, setting it to unbound.

Component references are managed by MC. All new test components are given a unique reference that was never used in the test campaign before (not even in a previous test case). The new numbers are increasing monotonously. The reference of the firstly created component is 3; the next one will be 4, and so on.

Using the value of an unbound component reference for anything will cause dynamic test case error.

Other operators (global functions):

```
boolean operator==(component component_value,
                   const COMPONENT& other_value); // Equal
boolean operator!=(component component_value,
                   const COMPONENT& other_value); // Not equal
```

6.3.14. Empty Types

Empty `record` and `set` types are not real built-in types in TTCN–3, but the C++ realization of these types also differs from regular records or sets. The empty types are almost identical to each other, only their names are different. That is why we treat them as predefined types.

Each empty type is defined in a C++ class, which is generated by the compiler. Using separate classes enables us to differentiate among them in C++ type polymorphism. For example, several empty types can be defined as incoming or outgoing types on the same TTCN–3 port type.

Let us consider the following TTCN–3 type definition as an example:

```
type record Dummy {};
```

The generated class will rely on an enumerated C type null_type, which is defined as follows:

```
enum null type {NULL VALUE };
```

The only possible value stands for the TTCN–3 empty record or array value (that is for "{}"), which is the only possible value of TTCN–3 type `Dummy`. Note that this type and value is also used in the definition of `record` of and `set of` type construct.

The generated C++ class `Dummy` will have the following member functions:

Table 25. Public member functions of the class `Dummy`

Member functions		**Notes**
Constructors	`Dummy()`	Initializes to unbound value.
	`Dummy(null type)`	Initializes to the only possible value.
	`Dummy(const Dummy&)`	Copy constructor.
Destructor	`~Dummy()`	
Assignment operators	`Dummy& operator=(null type)`	Assigns the only possible value and sets the bound flag.
	`Dummy& operator=(const Dummy&)`	
Comparison operators	`boolean operator==(Dummy) const`	Returns TRUE if both arguments are bound.
	`boolean operator==(const Dummy&) const`	
	`boolean operator!=(address) const`	Returns FALSE if both arguments are bound.
	`boolean operator!=(const Dummy&) const`	
Other member functions	`void log() const`	Puts the value, that is, {}, into log.
	`boolean is_bound() const`	Returns whether the value is bound.
	`void clean_up()`	Deletes the value, setting it to unbound.

Setting the only possible value is important, because using the value of an unbound variable for anything will cause dynamic test case error.

Other operators (global functions):

```
boolean operator==(null_type null_value, const Dummy& other_value);// Equal
boolean operator!=(null_type null_value, const Dummy& other_value);// Not equal
```

6.4. Compound Data Types

The user-defined compound data types are implemented in C++ classes. These classes are generated by the compiler according to type definitions. In contrast with the basic types, these classes can be found in the generated code.

6.4.1. Record and Set Type Constructs

The TTCN–3 type constructs `record` and `set` are mapped in an identical way to C++. There will be a C++ class for each record type in the generated code. This class builds up the record from its fields. [10] The fields can be either basic or compound types.

Let us consider the following example type definition. The types `t1` and `t2` can be arbitrary.

```
type record t3 {
  t1 f1,
  t2 f2
}
```

The generated class `t3` will have the following public member functions:

Table 26. Public member functions of the class `t3`

Member functions		Notes
Constructors	`t3()`	Initializes all fields to unbound value.
	`t3(const t1& par_f1, const t2& par_f2)`	Initializes from given field values. The number of arguments equals to the number of fields.
	`t3(const t3&)`	Copy constructor.
Destructor	`~t3()`	
Assignment operator	`t3& operator=(const t3&)`	Assigns the given value and setsthe bound flag for each field.
Comparison operators	`boolean operator==(const t3&) const`	Returns TRUE if all fields are equal and FALSE otherwise.
	`boolean operator!=(const t3&) const`	

Field access functions	`t1& f1(); t2& f2();`	Gives access to the first/second field.
	`const t1& f1() const; const t2& f2() const;`	The same, but it gives read-only access.
Other member functions	`int size_of() const`	Returns the size (number of fields).
	`void log() const`	Puts the value into log. Like { f1 := 5, f2 := ”abc”}.
	`boolean is_bound() const`	Returns whether the value is bound.
	`void clean_up()`	Deletes the value, setting it to unbound.

The record value is unbound if one or more fields of it are unbound. Using the value of an unbound variable for anything (even for comparison) will cause dynamic test case error.

Optional Fields in Records and Sets

TTCN–3 permits optional fields in record and set type definitions. An optional field does not have to be always present, it can be omitted. But the omission must be explicitly denoted. Let us change our last example to this.

```
type record t3 {
  t1 f1,
  t2 f2 optional
}
```

The optional fields are implemented using a C++ template class called `OPTIONAL` that creates an optional value from any type. In the definition of the generated class `t3` the type `t2` will be replaced by `OPTIONAL<t2>` everywhere and anything else will not be changed.

The instantiated template class `OPTIONAL<t2>` will have the following member functions:

Table 27. Table Public member functions of the class `OPTIONAL<t2>`

Member functions	Notes

Constructors	`OPTIONAL()`	Initializes to unbound value.
	`OPTIONAL(template_sel init_val)`	Initializes to omit value, if the argument is OMIT VALUE.
	`OPTIONAL(const t2& init_val)`	Initializes to given value.
	`OPTIONAL(const OPTIONAL& init_val)`	Copy constructor.
	`template <typename T_tmp>`	Initializes to given value of different (compatible) type.
	`OPTIONAL(const OPTIONAL<T_tmp>&)`	
	`template <typename T_tmp>`	Initializes to given optional value of different (compatible) type.
	`OPTIONAL(const T_tmp&)`	
Destructor	`~OPTIONAL()`	
Assignment operators	`OPTIONAL& operator=(template_sel)`	Assigns omit value, if the right value is OMIT VALUE.
	`OPTIONAL& operator=(const OPTIONAL&)`	Assigns the given optional value.
	`template <typename T_tmp>`	Assigns the given optional value of different (compatible) type.
	`OPTIONAL& operator=(const OPTIONAL<T_tmp>&)`	
	`template <typename T_tmp>`	Assigns the given value of different (compatible) type.
	`OPTIONAL& operator=(const T_tmp&)`	
Comparison operators	`boolean operator==(template_sel) const`	Returns TRUE if the value is omit and the right side is OMIT VALUE or FALSE otherwise.
	`boolean operator==(const OPTIONAL&) const`	Returns TRUE if the two values are equal or FALSE otherwise.
	`template <typename T_tmp>`	Returns TRUE if the two values of different (compatible) types are equal or FALSE otherwise.
	`boolean operator!=(template_sel) const`	
	`boolean operator!=(const OPTIONAL&) const`	
	`template <typename T_tmp>`	
	`boolean operator!=(const OPTIONAL<T_tmp>&) const`	

Casting operators	`operator t2&()`	Gives read-write access to the value. If the value was not previously present, sets the bound flag true and the value will be initialized to unbound.
	`operator const t2&() const`	Gives read-only access to the value. If the value is not present, causes a dynamic test case error.
Function call operators	`t2& operator()()`	Gives read-write access to the value. If the value was not previously present, sets the bound flag true and the value will be initialized to unbound.
	`const t2& operator()() const`	Gives read-only access to the value. If the value is not present, causes a dynamic test case error.
Other member functions	`boolean ispresent() const`	Returns TRUE if the value is present, FALSE if the value is omit or causes dynamic test case error if the value is unbound.
	`void log() const`	Puts the optional value into log. Either ”omit” or the value of t2.
	`boolean is_bound() const`	Returns whether the value is bound.
	`void clean_up()`	Deletes the value, setting it to unbound.

In some member functions of the template class `OPTIONAL` the enumerated C type `template_sel` is used. It has many possible values, but in the optional class only `OMIT_VALUE` can be used, which stands for the TTCN–3 omit. Usage of other predefined values of `template_sel` will cause dynamic test case error.

Using the value of an unbound optional field for anything will also cause dynamic test case error.

6.4.2. Union Type Construct

The TTCN–3 type construct union is implemented in a C++ class for each union type in the generated code. This class may contain any, but exactly one of its fields. The fields can be either basic or compound types or even identical types.

Let us consider the following example type definition. The types `t1` and `t2` can be arbitrary.

```
type union t3 {
  t1 f1,
  t2 f2
}
```

An ancillary enumerated type is created in the generated class `t3`, which represents the selection:

```
enum union_selection_type { UNBOUND_VALUE = 0, ALT_f1 = 1, ALT_f2 = 2 };
```

The type `t3::union_selection_type` is used to distinguish the fields of the union. The predefined constant values are generated as `t3::ALT_`<field name>.

The generated class `t3` will have the following public member functions:

Table 28. Public member functions of the class `t3`

Member functions		Notes
Constructors	`t3()`	Initializes to unbound value.
	`t3(const t3&)`	Copy constructor.
Destructor	`~t3()`	
Assignment operator	`t3& operator=(const t3&)`	Assigns the given value.
Comparison operators	`boolean operator==(const t3&) const`	Returns TRUE if the selections and field values are equal and FALSE otherwise.
	`boolean operator!=(const t3&) const`	
Field access functions	`const t1& f1() const`	Selects and gives access to the first field. If other field was previously selected, its value will be destroyed.
	`t1& f1()`	Gives read-only access to the first field. If other field is selected, this function will cause a dynamic test case error. So use get_selection() first.
	`t2& f2()`	
	`const t2& f2() const`	

Member functions		Notes
Other member functions	`union_selection_type get_selection() const`	Returns the current selection. It will return t3::UNBOUND VALUE if the value is unbound, t3::ALT_f1 if the first field was selected, and so on.
	`void log() const`	Puts the value into log. Example: { f1 := 5 } or { f2 := "abc" }.
	`boolean is_bound() const`	Returns whether the value is bound.
	`void clean_up()`	Deletes the value, setting it to unbound.

Using the value of an unbound `union` variable for anything will cause dynamic test case error.

The anytype

The TTCN-3 anytype is implemented as a C++ class named anytype. The class is generated only if an actual anytype access is present in the module. It has the same interface as any other C++ class generated for a union, with a few differences:

If a field is a built-in type or the address type, the name used in `union_selection_type` is the name of the runtime class implementing the type (usually the name of the type in all uppercase).

If a field is a user-defined type, the mapping rules in Mapping of Names and Identifiers above apply.

The names of field accessor functions are prefixed with AT_. This is necessary, because otherwise the accessor function looks like a constructor to C++.

For example, for the following module

```
module anyuser {
  type record myrec {}

  control {
    var anytype v_at;
  }
}
with {
  extension "anytype integer, myrec, charstring"
}
```

The generated class name will be "anytype". The union_selection_type enumerated type will be:

```
enum union_selection_type { UNBOUND_VALUE = 0, ALT_INTEGER = 1, ALT_myrec = 2,
ALT_CHARSTRING = 3 };
```

The field accessor methods will be:

```
INTEGER&    AT_INTEGER();
myrec&      AT_myrec();
CHARSTRING& AT_CHARSTRING();
```

6.4.3. Record of Type Construct

The TTCN–3 type construct `record` of makes a variable length sequence from one given type. This construct is implemented as a C++ class.

Let us consider the following example type definition. The type t1 can be arbitrary.

```
type record of t1 t2;
```

This definition will be translated to a C++ class that will be called t2.

There is an `enum` type called `null_type` defined in the Base Library that has only one possible value. NULL_VALUE stands for the empty `"record of"` value, that is, for {}.

Class `t2` will have the following public member functions:

Table 29. Public member functions of the class `t2`

Member functions		Notes
Constructors	`t2()`	Initializes to unbound value.
	`t2(null type)`	Initializes to the empty value.
	`t2(const t2&)`	Copy constructor.
Destructor	`~t2()`	
Assignment operator	`t2& operator=(null type)`	Assigns the empty value.
	`t2& operator=(const t2&)`	Assigns the given value.
Comparison operators	`boolean operator==(null type) const`	Returns TRUE if the two values are equal and FALSE otherwise.
	`boolean operator==(const t2&) const`	
	`boolean operator!=(null type) const`	
	`boolean operator!=(const t2&) const`	

Index operators	`t1& operator[](int)`	Gives access to the given element. Indexing begins from zero. If this element of the variable was never used before, new (unbound) elements will be allocated up to (and including) this index.
	`t1& opetator[](const INTEGER&)`	
	`const t1& operator[](int) const`	Gives read-only access to the given element. Index overflow causes dynamic test case error.
	`const t1& opetator[](const INTEGER&) const`	
Rotating operators	`t2 operator<⇐(int)`	C++ equivalent of operator <@. (rotate left)
	`t2 operator<⇐(const INTEGER&)`	
	`t2 operator>>=(int)`	C++ equivalent of operator @>. (rotate right)
	`t2 operator>>=(const INTEGER&)`	
Concatenation operator	`t2 operator+(const t2&) const`	Concatenates two arrays.

Other member functions	`int size_of() const`	Returns the number of elements, that is, the largest used index plus one and zero for the empty value.
	`void set_size(int new_size)`	Sets the number of elements to the given value. If the value has fewer elements new (unbound) elements are allocated at the end. The excess elements at the end are erased if the value has more elements than necessary.
	`t2 substr(int index, int returncount) const`	Returns the section of the array specified by the given start index and length.
	`t2 replace(int index, int len, const t2& repl) const`	Returns a copy of the array, where the section indicated by the given start index and length is replaced by the given array.
	`void log() const`	Puts the value into log. Like {1, 2, 3 }.
	`boolean is_bound() const`	Returns whether the value is bound.
	`void clean_up()`	Deletes the value, setting it to unbound.

A `record of` value is unbound if no value has been assigned to it or it has at least one unbound element. Using the value of an unbound `record of` variable for anything will cause dynamic test case error.

Starting with the largest index improves performance when filling a `record of value`.

Other operators (global functions):

```
boolean operator==(null_type null_value, const t2& other_value); // Equal
boolean operator!=(null_type null_value, const t2& other_value); // Not equal
```

Pre-generated `record of` and `set of` constructs

The C++ classes for the `record of` and `set of` constructs of most predefined TTCN-3 types are pre-generated and part of the TITAN runtime. Only a type alias (C++ `typedef`) is generated for instances of these types declared in TTCN-3 and ASN.1 modules. There is a class with regular memory allocation and one with optimized memory allocation pre-generated for each type. These classes are located in the `PreGenRecordOf` namespace.

Table 30. Pre-generated classes for `record of`/`set of` predefined types

C++ class name	Equivalent type in TTCN-3
PREGEN__RECORD__OF__BOOLEAN	record of boolean
PREGEN__RECORD__OF__INTEGER	record of integer
PREGEN__RECORD__OF__FLOAT	record of float
PREGEN__RECORD__OF__BITSTRING	record of bitstring
PREGEN__RECORD__OF__HEXSTRING	record of hexstring
PREGEN__RECORD__OF__OCTETSTRING	record of octetstring
PREGEN__RECORD__OF__CHARSTRING	record of charstring
PREGEN__RECORD__OF__UNIVERSAL__CHARSTRING	record of universal charstring
PREGEN__RECORD__OF__BOOLEAN__OPTIMIZED	record of boolean with { extension "optimize:memalloc" }
PREGEN__RECORD__OF__INTEGER__OPTIMIZED	record of integer with { extension "optimize:memalloc" }
PREGEN__RECORD__OF__FLOAT__OPTIMIZED	record of float with { extension "optimize:memalloc" }
PREGEN__RECORD__OF__BITSTRING__OPTIMIZED	record of bitstring with { extension "optimize:memalloc" }
PREGEN__RECORD__OF__HEXSTRING__OPTIMIZED	record of hexstring with { extension "optimize:memalloc" }
PREGEN__RECORD__OF__OCTETSTRING__OPTIMIZED	record of octetstring with { extension "optimize:memalloc" }
PREGEN__RECORD__OF__CHARSTRING__OPTIMIZED	record of charstring with { extension "optimize:memalloc" }
PREGEN__RECORD__OF__UNIVERSAL__CHARSTRING__OPTIMIZED	record of universal charstring with { extension "optimize:memalloc" }
PREGEN__SET__OF__BOOLEAN	set of boolean
PREGEN__SET__OF__INTEGER	set of integer
PREGEN__SET__OF__FLOAT	set of float
PREGEN__SET__OF__BITSTRING	set of bitstring
PREGEN__SET__OF__HEXSTRING	set of hexstring
PREGEN__SET__OF__OCTETSTRING	set of octetstring
PREGEN__SET__OF__CHARSTRING	set of charstring
PREGEN__SET__OF__UNIVERSAL__CHARSTRING	set of universal charstring
PREGEN__SET__OF__BOOLEAN__OPTIMIZED	set of boolean with { extension "optimize:memalloc" }
PREGEN__SET__OF__INTEGER__OPTIMIZED	set of integer with { extension "optimize:memalloc" }
PREGEN__SET__OF__FLOAT__OPTIMIZED	set of float with { extension "optimize:memalloc" }
PREGEN__SET__OF__BITSTRING__OPTIMIZED	set of bitstring with { extension "optimize:memalloc" }
PREGEN__SET__OF__HEXSTRING__OPTIMIZED	set of hexstring with { extension "optimize:memalloc" }

C++ class name	Equivalent type in TTCN-3
PREGEN__SET__OF__OCTETSTRING__OPTIMIZED	set of octetstring with { extension "optimize:memalloc" }
PREGEN__SET__OF__CHARSTRING__OPTIMIZED	set of charstring with { extension "optimize:memalloc" }
PREGEN__SET__OF__UNIVERSAL__CHARSTRING__OPTIMIZED	set ofs universal charstring with { extension "optimize:memalloc" }

6.4.4. `Set of` Type Construct

The `set of` construct of TTCN–3 is implemented similarly to `record of`. The external interface of this class is exactly the same as in case of `record of`. For more details please see the previous section.

In the internal implementation only the equality operator differs. Unlike in `record of`, it considers the unordered property of the `set of` type construct, that is, it returns `TRUE` if it is able to find exactly one pair for each element.

The index is a unique identifier for a `set of` element because the C++ class does not reorder the elements when a new element is added or an element is modified. The copy constructor also keeps the original order of elements.

6.4.5. Enumerated Types

The TTCN–3 `enumerated` type construct is implemented as a C++ class with an embedded enum type.

```
type enumerated Day { Monday (1), Tuesday, Wednesday (3) };
```

The example above will result in the following, very similar C `enum` type definition which is embedded in the C++ class `Day`:

```
enum enum_type { Monday = 1, Tuesday = 0, Wednesday = 3,
       UNKNOWN_VALUE = 2, UNBOUND_VALUE = 4 };
```

The automatic assignment of numeric values is done according to the standard. Note that there are two extra enumerated values in C, which stand for the unknown and unbound values. They are used in the conversion functions described below. The compiler assigns the smallest two non-negative integer numbers that are not used by the user-defined enumerated values to the unknown and unbound values.

When using the C `enum` type and its values from user code the names must be prefixed with the C++ class name. The `enum` type in the above example can be referenced with `Day::enum_type`, its values can be accessed as `Day::Monday`, `Day::Tuesday`, and so on.

The class `Day` will have the following public member functions:

Table 31. Public member functions of the class `Day`

Member functions		Notes
Constructors	`Day()`	Initializes to unbound value.
	`Day(int)`	Converts the given numeric value to Day::enum_type and initializes to it. Only valid values are accepted.
	`Day(enum_type)`	Initializes to a given value.
	`Day(const Day&)`	Copy constructor.
Destructor	`~Day()`	
Assignment operator	`Day& operator=(int)`	Converts the given numeric value to Day::enum_type and assigns it. Only valid values are accepted.
	`Day& operator=(enum_type)`	Assigns the given value.
	`Day& operator=(const Day&)`	
Comparison operators	`boolean operator==(enum_type) const`	Returns TRUE if the two values are equal and FALSE otherwise.
	`boolean operator==(const Day&) const`	
	`boolean operator!=(enum_type) const`	
	`boolean operator!=(const Day&) const`	
	`boolean operator<(enum_type) const`	
	`boolean operator<(const Day&) const`	
	`boolean operator<=(enum_type) const`	
	`boolean operator<=(const Day&) const`	
	`boolean operator>(enum_type) const`	
	`boolean operator>(const Day&) const`	
	`boolean operator>=(enum_type) const`	
	`boolean operator>=(const Day&) const`	
Casting operator	`operator enum_type() const`	Returns the enum_value.

Static conversion functions	`static const char *enum_to_str(enum_type)`	See below.
	`static enum_type str_to_enum(const char *)`	
	`static boolean is_valid_enum(int)`	
	`static int enum2int(enum_type)`	
	`static int enum2int(const Day&)`	
Non-static conversion functions	`int as_int() const`	See below
	`void from_int(int)`	
	`void int2enum(int)`	
Other member functions	`void log() const`	Puts the value into log. Like this: Monday
	`boolean is_bound() const`	Returns whether the value is bound.
	`void clean_up()`	Deletes the value, setting it to unbound.

The static member function `Day::enum_to_str` converts the given parameter of type `Day::enum_type` to a NULL terminated C character string. It returns the string "<unknown>", if the input is not a valid value of the TTCN–3 enumerated type. The returned string is read-only, it must not be modified.

The function `Day::str_to_enum` does the conversion in the reverse direction. It converts the symbolic enumerated identifier represented by a C character string back to the `Day::enum_type` equivalent. It returns the value `Day::UNKNOWN_VALUE` if the input string is not the equivalent of any of the possible values in the enumerated type. The behavior of this function is undefined if the input parameter does not point to an addressable memory area.

In the above two functions the strings are treated case sensitive and they shall not contain any whitespace or other characters that are not part of the enumerated value. In case of ASN.1 `ENUMERATED` types the strings used by `enum_to_str`, `str_to_enum` and log represent the TTCN–3 view of the enumerated value, that is, the hyphenation characters are mapped to a single underscore character. For example, if an ASN.1 enumerated type has a value with name `my-enum-value` and numeric value 2, the function `enum_to_str` will return the string `"my_enum_value"` if the input parameter equals to 2. Of course, its C++ equivalent will be `my_enum_value` with numeric value 2.

Static member function `Day::is_valid_enum` returns the Boolean value `TRUE` if there is a defined enumerated value having numeric value equal to the `int` parameter and `FALSE` otherwise.

The static member function `Day::enum_to_int` converts the given parameter of type Day or `Day::enum_type` to its numeric value. The member function `as_int` does the same thing for the enumerated instance.

The member function `int_to_enum` initializes the enumerated instance with the enumerated value having numeric value equal to the given `int` parameter. A dynamic test case error is displayed if

there is no such enumerated value. The member function `from_int` does the same thing.

If a value of type `int` is passed to the constructor or assignment operator the value is accepted only if it is a numerical representation of a valid enumerated value, that is, the function `is_valid_enum` returns `TRUE`. A dynamic test case error occurs otherwise.

To avoid run-time errors at the decoding of invalid messages the Test Port writer should use the constructor or assignment operator in this way:

```
Day myDayVar;
int myIntVar = buffer[position];
if (Day::is_valid_enum(myIntVar)) myDayVar = myIntVar;
else myDayVar = Day::UNKNOWN_VALUE;
```

Using the value of an unbound enumerated variable for anything will cause dynamic test case error.

6.4.6. The `address` Type

The special TTCN–3 data type `address` is represented in C++ as if it was a regular data type. The name of the equivalent C++ class is `ADDRESS`. If it is an alias to another (either built-in or user-defined) type then a C++ `typedef` is used.

6.4.7. The TTCN-3 open type

The special TTCN–3 type `any` (i.e. the TTCN-3 open type) is represented in C++ by the structure `ANY_TYPE`, and `template any` is represented by the structure `ANY_TEMPLATE_TYPE`.

These types can only appear as the types of external function parameters. They have the following declarations:

```
struct ANY_TYPE {
  const char* name;
  void* data;
};

struct ANY_TEMPLATE_TYPE {
  const char* name;
  void* data;
};
```

The `name` field identifies the type of the actual parameter. It contains the name of the type as a string. In case of built-in types, this is the name of the type in TTCN-3. In case of user-defined types, the name of the type is prefixed by `@<module_name>.`.

The `data` field contains the address of the actual parameter. It needs to be converted to the actual parameter's type before it can be accessed (based on the contents of the `name` field).

Both fields are automatically set by the generated code before the external function is called.

Usage example:

```
INTEGER MyExternalFunction(const ANY_TYPE& p__data)
{
  if (strcmp(p__data.name, "integer") == 0) {
    return (*(INTEGER*)(p__data.data));
  }
  else if (strcmp(p__data.name, "@MyModule.MyRecordType") == 0) {
    return ((MyRecordType*)(p__data.data))->myIntegerField();
  }
  ...
}
```

6.5. Predefined Functions

Annex C of Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 1: Core Language European Telecommunications Standards and Annex B of Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 7: Using ASN.1 with TTCN–3 European Telecommunications define a couple of predefined functions. Most of them perform conversion between the built-in types of TTCN–3. In our test executor these functions are implemented in the Base Library in C++ language. They are available not only in TTCN–3 , but they can be called directly from Test Ports as well.

The prototypes for these functions can be found in `$TTCN3_DIR/include/Addfunc.hh`, but for easier navigation we list them also in the present document.

The majority of these functions have more than one polymorphic version: when appropriate, one of them takes literal (built-in) C++ types as arguments instead of the objects of equivalent C++ classes. For instance, if the incoming argument is stored in an `int` variable in your C++ code, you should not construct a temporary object of class `INTEGER` because passing an `int` is faster and produces smaller binary code. Similarly, the returned type is also literal when it is possible.

6.5.1. `Integer` to character

```
extern CHARSTRING int2char(int value);
extern CHARSTRING int2char(const INTEGER& value);
```

6.5.2. Character to `integer`

```
extern int char2int(char value);
extern int char2int(const char *value);
extern int char2int(const CHARSTRING& value);
```

6.5.3. Integer to universal character

```
extern UNIVERSAL_CHARSTRING int2unichar(int value);
extern UNIVERSAL_CHARSTRING int2unichar(const INTEGER& value);
```

6.5.4. Universal character to integer

```
extern int unichar2int(const universal_char& value);
extern int unichar2int(const UNIVERSAL_CHARSTRING& value);
```

6.5.5. Bitstring to integer

```
extern INTEGER bit2int(const BITSTRING& value);
```

6.5.6. Hexstring to integer

```
extern INTEGER hex2int(const HEXSTRING& value);
```

6.5.7. Octetstring to integer

```
extern INTEGER oct2int(const OCTETSTRING& value);
```

6.5.8. Charstring to integer

```
extern INTEGER str2int(const char *value);
extern INTEGER str2int(const CHARSTRING& value);
```

6.5.9. Integer to bitstring

```
extern BITSTRING int2bit(const INTEGER& value, const INTEGER& length);
```

6.5.10. Integer to hexstring

```
extern HEXSTRING int2hex(const INTEGER& value, const INTEGER& length);
```

6.5.11. Integer to octetstring

```
extern OCTETSTRING int2oct(const INTEGER& value, const INTEGER& length);
```

6.5.12. `Integer` to `charstring`

```
extern CHARSTRING int2str(int value);
extern CHARSTRING int2str(const INTEGER& value);
```

6.5.13. Length of string Type

This function is built into the equivalent C++ classes of all TTCN–3 string types:

```
int <any_string_type>::lengthof() const;
```

6.5.14. Number of elements in a structured type

This function is built into the C++ template classes of `record of` and `set of` types:

```
int <any_record_of_or_set_of_type>::size_of() const;
```

This function is currently not implemented for `record` and `set` types.

6.5.15. The `IsPresent` Function

This function is built into the wrapper C++ template class `OPTIONAL`:

```
boolean <any_optional_field>::ispresent() const;
```

6.5.16. The `IsChosen` Function

These functions are built into the equivalent C++ classes of TTCN–3 union types:

```
boolean <union_type>::ischosen(
<union_type>::union_selection_type checked_selection) const;
```

6.5.17. The `regexp` Function

```
extern CHARSTRING regexp(const CHARSTRING& instr,
const CHARSTRING& expression, const INTEGER& groupno);
```

6.5.18. Bitstring to charstring

```
extern CHARSTRING bit2str(const BITSTRING& value);
```

6.5.19. Hexstring to charstring

```
extern CHARSTRING hex2str(const HEXSTRING& value);
```

6.5.20. Octetstring to character string

```
extern CHARSTRING oct2str(const OCTETSTRING& value);
```

6.5.21. Character string to octetstring

```
extern OCTETSTRING str2oct(const char *value);
extern OCTETSTRING str2oct(const CHARSTRING& value);
```

6.5.22. Bitstring to hexstring

```
extern HEXSTRING bit2hex(const BITSTRING& value);
```

6.5.23. Hexstring to octetstring

```
extern OCTETSTRING hex2oct(const HEXSTRING& value);
```

6.5.24. Bitstring to octetstring

```
extern OCTETSTRING bit2oct(const BITSTRING& value);
```

6.5.25. Hexstring to bitstring

```
extern BITSTRING hex2bit(const HEXSTRING& value);
```

6.5.26. Octetstring to hexstring

```
extern HEXSTRING oct2hex(const OCTETSTRING& value);
```

6.5.27. `Octetstring` to `bitstring`

```
extern BITSTRING oct2bit(const OCTETSTRING& value);
```

6.5.28. `Integer` to `float`

```
extern double int2float(int value);
extern double int2float(const INTEGER& value);
```

6.5.29. `Float` to `integer`

```
extern INTEGER float2int(double value);
extern INTEGER float2int(const FLOAT& value);
```

6.5.30. The Random Number Generator Function

The implementation is based on functions `srand48` and `drand48` of `libc`.

```
extern double rnd();
extern double rnd(double seed);
extern double rnd(const FLOAT& seed);
```

6.5.31. The Substring Function

Implemented for all string types.

```
extern BITSTRING substr(const BITSTRING& value, const INTEGER& index,
  const INTEGER& returncount);
extern HEXSTRING substr(const HEXSTRING& value, const INTEGER& index,
  const INTEGER& returncount);
extern OCTETSTRING substr(const OCTETSTRING& value, const INTEGER& index,
  const INTEGER& returncount);
extern CHARSTRING substr(const CHARSTRING& value, const INTEGER& index,
  const INTEGER& returncount);
extern UNIVERSAL_CHARSTRING substr(const UNIVERSAL_CHARSTRING& value,
  const INTEGER& index, const INTEGER& returncount);
```

6.5.32. Character string to `float`

```
extern double str2float(const char *value);
extern double str2float(const CHARSTRING& value);
```

6.5.33. The Replace Function

Implemented for all string types.

```
extern BITSTRING replace(const BITSTRING& value, const INTEGER& index,
  const INTEGER& len, const BITSTRING& repl);
extern HEXSTRING replace(const HEXSTRING& value, const INTEGER& index,
  const INTEGER& len, const HEXSTRING& repl);
extern OCTETSTRING replace(const OCTETSTRING& value, const INTEGER& index,
  const INTEGER& len, const OCTETSTRING& repl);
extern CHARSTRING replace(const CHARSTRING& value, const INTEGER& index,
  const INTEGER& len, const CHARSTRING& repl);
extern UNIVERSAL_CHARSTRING replace(const UNIVERSAL_CHARSTRING& value,
  const INTEGER& index, const INTEGER& len, const UNIVERSAL_CHARSTRING& repl);
```

6.5.34. Octetstring to character string

```
extern CHARSTRING oct2char(const OCTETSTRING& value);
```

6.5.35. Character string to octetstring

```
extern OCTETSTRING char2oct(const char *value);
extern OCTETSTRING char2oct(const CHARSTRING& value);
```

6.5.36. The `Decompose` Function

Not implemented yet.

6.5.37. Additional Non-Standard Functions

```
extern BITSTRING str2bit(const char *value);
extern BITSTRING str2bit(const CHARSTRING& value);
extern HEXSTRING str2hex(const char *value);
extern HEXSTRING str2hex(const CHARSTRING& value);
extern CHARSTRING float2str(double value);
extern CHARSTRING float2str(const FLOAT& value);

template<typename TTCN_TYPE>
CHARSTRING ttcn_to_string(const TTCN_TYPE& ttcn_data)

template<typename TTCN_TYPE>
void string_to_ttcn(const CHARSTRING& ttcn_string, TTCN_TYPE& ttcn_value)

extern UNIVERSAL_CHARSTRING oct2unichar(const OCTETSTRING& invalue);
extern UNIVERSAL_CHARSTRING oct2unichar(const OCTETSTRING& invalue,
  const CHARSTRING& string_encoding);

extern OCTETSTRING unichar2oct(const UNIVERSAL_CHARSTRING& invalue);
extern OCTETSTRING unichar2oct(const UNIVERSAL_CHARSTRING& invalue,
  const CHARSTRING& string_encoding);

extern CHARSTRING get_stringencoding(const OCTETSTRING& encoded__value);
extern OCTETSTRING remove_bom(const OCTETSTRING& encoded__value);

extern CHARSTRING encode_base64(const OCTETSTRING& msg, bool use_linebreaks);
extern CHARSTRING encode_base64(const OCTETSTRING& msg);
extern OCTETSTRING decode_base64(const CHARSTRING& b64);
```

See the section "Additional predefined functions" in the Programmer"s Technical Reference for more details.

6.6. Using the Signature Classes

A Test Port has three outgoing and three incoming types of operation that require the usage of signatures. These are `call` (`getcall`), `reply` (`getreply`) and `raise` (`catch`). Because of this, there are three representation formats (classes generated by the compiler) of a signature the Test Port writer should be familiar with. This section describes these classes using an example.

Let us suppose the following signature definition:

```
signature MyProc(in integer inPar, out float outPar,
    inout bitstring inoutPar)
        return hexstring
        exception(charstring, integer, boolean);
```

The classes generated and needed to write a Test Port using this signature are `MyProc_call`, `MyProc_reply` and `MyProc_exception`. These represent the parameters, the return value and the

exception type and value of the signature needed by a call, reply or raise.

For example, if a port uses the signature `MyProc` as an output remote procedure, the Test Port gets the outgoing parameters for a call operation towards the system in an instance of the class `MyProc_call`. In this case the classes `MyProc_reply` and `MyProc_exception` are used for placing an incoming reply or raise operation in the queue of the port (using the functions `incoming_reply` and `incoming_exception` of the port class).

6.6.1. The Representation of the Input Parameters

The class `MyProc_call` (using the above example) represents all incoming parameters of the signature `MyProc`. It temporary stores the parameters inPar and inoutPar.

The generated class `MyProc_call` will have the following public member functions:

Table 32. Public member functions of the class `MyProc_call`

Member functions		**Notes**
Parameter access functions	`INTEGER& inPar()`	Gives access to parameter inPar.
	`const INTEGER& inPar() const`	
	`BITSTRING& inoutPar()`	The same, but it gives read-only access.
	`const BITSTRING& inoutPar() const`	
Other member functions	`void log() const`	Puts the parameters into log.

The parameters can be accessed via their access functions that have the same names as the parameters (name mapping also applies to these functions).

6.6.2. The Output Parameters and Return Value

The output parameters and return value (if defined) are represented by the class `MyProc_reply` that has the following public member functions:

Table 33. Public member functions of the class `MyProc_reply`

Member functions		**Notes**
Parameter access functions	`FLOAT& outPar()const FLOAT& outPar() const`	Gives access to parameter outPar.
	`BITSTRING& inoutPar() const BITSTRING& inoutPar() const`	The same, but it gives read-only access.
Access function for return value	`HEXSTRING& return value()`	Gives access to the return value.
	`const HEXSTRING& return value() const`	
Other member functions	`void log() const`	Puts the parameters into log.

The parameters can be accessed by their access functions, and the return value can be accessed via the function `return_value()`.

6.6.3. Representation of Signature Exceptions

The class representing the exceptions of a signature (remote procedure) is similar to the representation of the union data type. Using the above example this class is called `MyProc_exception`. This class is generated only if the signature has at least one exception type.

Table 34. Public member functions of the class `MyProc_exception`

Member functions		**Notes**
Constructors	`MyProc_exception()`	Initializes to unbound value.
	`MyProc_exception(const MyProc_exception&)`	Copy constructor.
Destructor	`~MyProc_exception()`	
Assignment operator	`MyProc_exception& operator=(const MyProc_exception&)`	Assigns the given value.
Field access functions	`CHARSTRING& CHARSTRING_field()`	Selects and gives access to the CHARSTRING field. If other field was previously selected, its value will be destroyed.
	`const CHARSTRING&CHARSTRING_field() const`	Gives read-only access to the CHARSTRING field. If other field is selected, this function will cause dynamic test case error. So use get selection() first.
	`INTEGER& INTEGER_field() const`	
	`BOOLEAN& BOOLEAN_field() const`	
Other member functions	`MyProc_exception::exception_selection_type get_selection() const`	Returns the current selection. It will return MyProc exception::UNBOUND VALUE if the exception is unbound, MyProc exception::ALT CHARSTRING if a charstring value is present in the exception, and so on.
	`void log() const`	Puts the contents of the exception into the log.

If an exception type is a user-defined type the field name will be constructed from the C++ namespace name of the module that the exception type resides in and the name of the C++ class that realizes the exception type. The two identifiers are glued together using a single underscore character. Please note that the namespace name is always present in the identifiers, even if the exception type is defined in the same module as the signature.

For example, if exception type `My_Record` is defined in module `My_Module` the respective field access functions will be named as `My__Module_My__Record_field` and the associated enum value will be `MyProc_exception::ALT_My__Module_My__Record`.

6.7. Object references

All function parameters, return values and variables of class type must use the `OBJECT_REF` wrapper class, with the class type as its template parameter (e.g. `MyClass` in TTCN-3 translates to `OBJECT_REF<MyClass>` in C++). This class contains a pointer to the class object itself, and handles its reference counting and deallocation.

Accessing members and methods through `OBJECT_REF` is done through the operator `->` (as if it were a pointer in C++). Methods of `OBJECT_REF` are accessed through the operator `.` (see below).

Table 35. Public methods of the template class `OBJECT_REF<T>`, where `T` is a class type

Member functions		**Notes**
Constructors	`OBJECT_REF()`	Initializes to `null` reference.
	`OBJECT_REF(null_type)`	Initializes to `null` reference.
	`OBJECT_REF(T*)`	Initializes to a new object, which needs to be allocated with the C++ operator `new`.
	`OBJECT_REF(const OBJECT_REF<T>&)`	Copy constructor.
	`template <typename T2> OBJECT_REF(OBJECT_REF<T2>&)`	Copy constructor for a reference of a different class type, referring to an object of type `T` or a subclass of `T`.
Destructor	`~OBJECT_REF()`	
Assignment operators	`OBJECT_REF& operator=(null_type)`	Assigns the `null` reference.
	`OBJECT_REF& operator=(const OBJECT_REF<T>&)`	Assigns a reference of `T`.
	`template <typename T2> OBJECT_REF& operator=(OBJECT_REF<T2>&)`	Assigns a reference of a different class type, referring to an object of type `T` or a subclass of `T`.

Comparison operators	`boolean operator==(null_type) const`	Returns TRUE if the reference is `null`.
	`boolean operator!=(null_type) const`	Returns FALSE if the reference is `null`.
	`boolean operator==(const OBJECT_REF<T>&) const`	Returns TRUE if both this and the parameter refer to the same object.
	`boolean operator!=(const OBJECT_REF<T>&) const`	Returns FALSE if both this and the parameter refer to the same object.
Access operators	`T* operator*()`	Returns a pointer to the actual object.
	`T* operator->()`	Accesses the object's members and methods.
	`const T* operator->() const`	Accesses the object's constant members and methods.
Other methods	`void log() const`	Puts the object (or `null`) into the log.
	`boolean is_bound() const`	Returns TRUE if the reference is not `null`.
	`boolean is_value() const`	Returns TRUE if the reference is not `null`.
	`boolean is_present() const`	Returns TRUE if the reference is not `null`.

[9] The built-in `verdict` and `boolean` constants in TTCN–3 shall be written with all lowercase letters, such as true or pass. Although previous compiler versions have accepted `TRUE` or `PASS` as well, these words are treated by the compiler as regular identifiers as specified in the standard.

[10] This section deals with the record and set types that have at least one field. See Empty Types for the C++ mapping of empty record and set types.

Chapter 7. Tips & Troubleshooting

Information not fitting in any of the previous chapters is given in this chapter.

7.1. Migrating Existing C++ Code to the Naming Rules of Version 1.7

When using the new naming rules[11] the compiler generates a C++ namespace for each TTCN–3 and ASN.1 module. The name of the namespace corresponds to the module. The generated C++ entities of a module are all placed in its namespace; therefore all the test port or protocol module code must use these namespaces.

Rules to follow when writing C++ code:

- When referencing an entity located in a different module its C++ name has to be prefixed with the namespace name of that module.
- A test port class must be placed into the namespace of its module.
- Encoding and decoding functions must be placed into the namespace of the TTCN–3 module in which the external function was defined.
- All C++ entities have to be placed into namespace. An exception to this may be C++ entities used only locally; these are defined with the keyword `static`.
- For convenience the `using namespace` directive can be used in C++ source files. It is forbidden to use this directive in header files!
- C++ enum types are placed in the scope of their value class; enum types have to be prefixed by the C++ name of the value class.[12]

7.2. Using External C++ Functions in TTCN–3 Test Suites

Sometimes standard library functions[13] are called in the test suite or there is a need for efficiently implemented "bit-crunching" functions in the TTCN–3 ATS. In these cases functions to be called from the test suite can be developed in C++.

There are the standard library functions as well as other libraries in the C++ functions. The logging and error handling facilities of the run-time environment are also available as in case of Test Ports.

Since version 1.4.pl1 the semantic analyzer of the compiler checks the import statements thoroughly. Therefore one cannot use the virtual C++ modules as before: C++ functions must be defined as external functions to be accessible from TTCN–3 modules.

For example, the following definitions make two C++ functions accessible from TTCN–3 module `MyExample` and from any other module that imports `MyExample`.

7.2.1. Example TTCN–3 Module (MyExample.ttcn)

```
module MyExample {
[...]
  external function MyFunction(integer par1, in octetstring par2)
    return bitstring;
  external function MyAnotherFunction(inout My_Type par1,
    out MyAnotherType par2);
[...]
}
```

The compiler will translate those external function definitions to C++ function prototypes in the generated header file `MyExample.hh`:

```
[...]
  extern BITSTRING MyFunction(const INTEGER& par1, const OCTETSTRING& par2);
  extern void MyAnotherFunction(My__Type& par1, MyAnotherType& par2);
[...]
```

Both pre-defined and user-defined TTCN–3 data types can be used as parameters and/or return types of the C++ functions. The detailed description of the equivalent C++ classes as well as the name mapping rules are described in chapter XML Encoding (XER).

Using templates as formal parameters in external functions is possible, but not recommended because the API of the classes realizing templates is not documented and subject to change without notice.

The formal parameters of external TTCN–3 functions are mapped to C++ function parameters according to the following table:

Table 35. TTCN–3 formal parameters and their C++ equivalents

TTCN–3 formal parameter	Its C++ equivalent
`[in] MyType myPar`	`const MyType& myPar`
`out MyType myPar`	`MyType& myPar`
`inout MyType myPar`	`MyType& myPar`
`[in] template MyType myPar`	*Not recommended.*

NOTE In versions 1.6.pl3 and earlier the in keyword had an extra meaning in formal parameter lists. According to the TTCN–3 standard the parameter definitions `MyType myPar` and in `MyType  myPar` are totally equivalent, but the earlier versions of the compiler distinguished them. Unless the keyword `in` was present the compiler passed the parameter by value (involving a copy constructor call) instead of using a const reference. That is why it was recommended to use an explicit in keyword in parameter lists of external functions.

Due to the strictness of the TTCN–3 semantic analyzer one cannot use C/C++ data types with

external functions as formal parameters or return types, only TTCN–3 and ASN.1 data types are allowed. Similarly, one cannot use pointers as parameters or return values because they have no equivalents in TTCN–3 .

The external functions can be implemented in one or more C++ source files. The generated header file that contains the prototypes of the external functions shall be included into each C++ source file. This file makes accessible all built-in data types, the user-defined types of the corresponding TTCN–3 module and all available services of the run-time environment (logging, error handling, etc.).

The name, return type and the parameters of the implemented C++ functions must match exactly the generated function prototypes or the compilation will fail. The generated function prototype is in the namespace of the module, therefore the implementation of the function has to be placed in that namespace, too.

7.3. Logging in Test Ports or External Functions

When developing Test Ports or external functions the need may arise for debug messages. Instead of using `printf` or `fprintf`, there is a simple way to put these messages into the log file of test executor. This feature can be also useful in case when an error or warning situation is encountered in the Test Port, especially when decoding an incoming message.

There is a class called `TTCN_Logger` in the Base Library, which takes care of logging. For historical reasons it has a static instance (object), which is called `TTCN_logger`. Since all member functions of `TTCN_Logger` are static, they can be and should be called without the logger object. The usage of object `TTCN_logger` should be avoided in newly written code.

The class `TTCN_Logger` provides some public member functions. Using them any kind of message can be put into the log file. There are two ways to log a single message, the unbuffered and the buffered mode.

7.3.1. Unbuffered Mode

In unbuffered mode the message will be put into log immediately as a separate line together with a time stamp. Thus, the entire message must be passed to the logger class at one function call. The log member function of the logger class should be used. Its prototype is:

```
static void TTCN_Logger::log(int severity, const char *fmt, ⋯);
```

The parameter severity is used for filtering the log messages. The allowed values of the parameter are listed in table "First level (coarse) log filtering" in the Programmer's Technical Reference. We recommend using in Test Ports only `TTCN_WARNING`, `TTCN_ERROR` and `TTCN_DEBUG`. The parameter `fmt` is a pointer to a format string, which is interpreted as in `printf(3)`. The dots represent the optional additional parameters that are referred in format string. There is no need to put a newline character at the end of format string; otherwise the log file will contain an empty line after your entry.

Here is an example, which logs an integer value:

```
int myVar = 5;
TTCN_Logger::log(TTCN_WARNING, ``myVar = %d'', myVar);
```

Sometimes the string to be logged is static. In such cases there is no need for `printf`-style argument processing, which may introduce extra risks if the string contains the character `%`. The logger class offers a function for logging a static (or previously assembled) string:

```
static void TTCN_Logger::log_str(int severity, const char *str);
```

The function `log_str` runs significantly faster than log because it bypasses the interpretation of the argument string.

There is another special function for unbuffered mode:

```
static void TTCN_Logger::log_va_list(int severity, const char *fmt,
    va_list ap);
```

The function `log_va` list resembles to log, but it takes the additional `printf` arguments in one va_list structure; va_list is defined in the standard C header file `stdarg.h` and used in functions with variable number of arguments.

This function (and especially its buffered mode version, `log_event_va_list`) is useful if there is a need for a wrapper function with `printf`-like syntax, but the message should be passed further to `TTCN_Logger`. With these functions one can avoid the handling of temporary buffers, which could be a significant performance penalty.

7.3.2. Buffered Mode

As opposite to the unbuffered operation, in buffered mode the logger class stores the message fragments in a temporary buffer. New fragments can be added after the existing ones. When finished, the fragments can be flushed after each other to the log file as a simple message. This mode is useful when assembling the message in many functions since the buffer management of logger class is more efficient than passing the fragments as parameters between the functions.

In buffered mode, the following member functions are available.

begin_event

`begin_event` creates a new empty event buffer within the logger. You have to pass the severity value, which will be valid for all fragments (the list of possible values can be found in the table "First level (coarse) log filtering" in the Technical Reference. If the logger already has an unfinished event when begin event is called the pending event will be pushed onto an internal stack of the logger. That event can be continued and completed after finishing the newly created event.

```
static void TTCN_Logger::begin_event(int severity);
```

log_event

`log_event` appends a new fragment at the end of current buffer. The parameter `fmt` contains a `printf` format string like in unbuffered mode. If you try to add a fragment without initializing the buffer by calling begin event, your fragment will be discarded and a warning message will be logged.

```
static void TTCN_Logger::log_event(const char *fmt, ...);
```

log_char

`log_char` appends the character c at the end of current buffer. Its operation is very fast compared to `log_event`.

```
static void TTCN_Logger::log_char(char c);
```

log_event_str and log_event_va_list

The functions `log_str` and `log_va_list` also have the buffered versions called `log_event_str` and `log_event_va_list`, respectively. Those interpret the parameters as described in case of unbuffered mode.

```
static void TTCN_Logger::log_event_str(const char *str);
static void TTCN_Logger::log_event_va_list(const char *fmt, va_list ap);
```

OS_error

The function `OS_error` appends the textual description of the error code stored in global variable `errno` at the end of current buffer. Thereafter that variable `errno` will be set to zero. The function does nothing if the value of `errno` is already zero. For further information about possible error codes and their textual descriptions please consult the manual page of `errno(3)` and `strerror(3)`.

```
static void TTCN_Logger::OS_error();
```

log

The C++ classes of predefined and compound data types are equipped with a member function called `log`. This function puts the actual value of the variable at the end of current buffer. Unbound variables and fields are denoted by the symbol `<unbound>`. The contents of TTCN–3 value objects can be logged only in buffered mode.

```
void <any TTCN-3 type>::log() const;
```

end_event

The function `end_event` flushes the current buffer into the log file as a simple message, then it destroys the current buffer. If the stack of pending events is not empty the topmost event is popped from the stack and becomes active. The time stamp of each log entry is generated at the end and not at the beginning. If there is no active buffer when `end_event` is called, a warning message will be logged.

```
static void TTCN_Logger::end_event();
```

If an unbuffered message is sent to the logger while the buffer contains a pending event the unbuffered message will be printed to the log immediately and the buffer remains unchanged.

7.3.3. Logging Format of TTCN-3 Values and Templates

TTCN-3 values and templates can be logged in the following formats:

TITAN legacy logger format: this is the default format which has always been used in TITAN

TTCN-3 format: this format has ttcn-3 syntax, thus it can be copied into TTCN-3 source files.

Differences between the formats:

Value/template	Legacy format output	TTCN-3 format output
Unbound value	"<unbound>"	"-"
Uninitialized template	"<uninitialized template>"	"-"
Enumerated value	name (number)	name

The "-" symbol is the NotUsedSymbol which can be used inside compound values, but when logging an unbound value which is not inside a record or record of the TTCN-3 output format of the logger is actually not a legal TTCN-3 value/template because a value or template cannot be set to be unbound. Thus this output format can be copy-pasted from a log file into a ttcn-3 file or to a module parameter value in a configuration file only if it semantically makes sense.

The C++ API extensions to change the logging format:
A new enum type for the format in TTCN_Logger class:+ `enum data_log_format_t { LF_LEGACY, LF_TTCN }`;
Static functions to get/set the format globally:
`data_log_format_t TTCN_Logger::get_log_format();void TTCN_Logger::set_log_format(data_log_format_t p_data_log_format)`;
A helper class to use a format until the end of the scope, when used as local variable. This can be used as follows:

```
{
    Logger_Format_Scope lfs(TTCN_Logger::LF_TTCN); // sets TTCN-3 log format
    <log some values and templates>
} // end of scope -> the original format is restored
```

It is recommended to use this helper class because using directly the format setting functions of `TTCN_Logger` is more error prone, if the globally used logging format is not restored properly then log files might contain values/templates in a mixed/unexpected format.

7.3.4. Logging sensitive data

Values and templates of top level types marked as sensitive data (i.e. types with the `sensitive_data` extension attribute) are not automatically redacted, when they are logged from user C++ code.

They can be redacted manually using the `TTCN_Logger::begin_sensitive_data` and `TTCN_Logger::end_sensitive_data` functions. The `.log` function of the value or template should be preceded by `TTCN_Logger::begin_sensitive_data` and should be followed by `TTCN_Logger::end_sensitive_data`.

Alternatively these two `TTCN_Logger` functions can be used to redact any part of a log the user deems to be sensitive.

Record fields and array elements marked as sensitive data are automatically redacted by their structure's generated C++ classes.

The redacting of sensitive data can be turned off with the `TTCN_Logger::ignore_sensitive_data` function. This has no ending counterpart, and lasts until the end of the current logging event.

7.3.5. Examples

The example below demonstrates the combined usage of buffered and unbuffered modes as well as the working mechanism of the event stack:

```
TTCN_Logger::begin_event(TTCN_DEBUG);
TTCN_Logger::log_event_str("first ");
TTCN_Logger::begin_event(TTCN_DEBUG);
TTCN_Logger::log_event_str("second ");
TTCN_Logger::log_str(TTCN_DEBUG, "third message");
TTCN_Logger::log_event_str("message");
TTCN_Logger::end_event();
TTCN_Logger::log_event_str("message");
TTCN_Logger::end_event();
```

The above code fragment will produce three lines in the log in the following order:

`third message second message first message`

If the code calls a C++ function that might throw an exception while the logger has an active event

buffer care must be taken that event is properly finished during stack unwinding. Otherwise the stack of the logger and the call stack of the program will get out of sync. The following example illustrates the proper usage of buffered mode with exceptions:

```
TTCN_Logger::begin_event(TTCN_DEBUG);
try {
  TTCN_Logger::log_event_str("something");
  // a function is called from here
  // that might throw an exception (for example TTCN_error())
  TTCN_Logger::log_event_str("something else");
  TTCN_Logger::end_event();
} catch (...) {
  // don't forget about the pending event
  TTCN_Logger::end_event();
  throw;
}
```

Example for redacting a sensitive integer variable:

```
TTCN_Logger::begin_event(TTCN_USER);
TTCN_Logger::log_event_str("my_int_var := ");
TTCN_Logger::begin_sensitive_data();
my_int_var.log();
TTCN_Logger::end_sensitive_data();
// maybe log other stuff aswell
TTCN_Logger::end_event();
```

7.4. Error Recovery during Test Execution

If a fatal error is encountered in the Test Port, you should call the function `TTCN_error` must be called to do the error handling. It has the following prototype in the Base Library:

```
void TTCN_error(const char *fmt, ...);
```

The parameter `fmt` contains the reason of the error in a NUL terminated character string in the format of a `printf` format string. If necessary, additional values should be passed to `TTCN_error` as specified in the format string. The error handling in the executable test program is implemented using C++ exceptions so the function `TTCN_error` never returns; instead, it throws an exception. The exception value contains an instance of the empty class called `TC_Error`. This exception is normally caught at the end of each test case and module control part. After logging the reason `TTCN_Logger::OS error()` is called. Finally, the verdict is set to error and the test executor performs an error recovery, so it continues the execution with the next test case.

It is not recommended to use own error recovery combined with the default method (that is, catching this exception).

7.5. Using UNIX Signals

The UNIX signals may interrupt the normal execution of programs. This may happen when the program executes system calls. In this case, when the signal handler is finished the system call will fail and return immediately with an error code.

In the executable test program there are system calls not only in the Base Library, but in Test Ports as well. Since the other Test Ports that you are using may have been written by many developers, one cannot be sure that they are prepared to the effects of signals. So it is recommended to avoid using signals in Test Ports.

7.6. Mixing C and C++ Modules

Modules written in C language may be used in the Test Ports. In this case the C header files must be included into the Test Port source code and the object files of the C module must be linked to the executable. Using a C compiler to compile the C modules may lead to errors when linking the modules together. This is because the C and C++ compilers use different rules for mapping function names to symbol names of the object file to avoid name clashes caused by the C++ polymorphism. There are two possible solutions to solve this problem:

1. Use the same C++ compiler to compile all of your source code (including C modules).
2. If the first one is impossible (when using a third party software that is available in binary format only), the definitions of the C header file must be put into an `extern "C"` block like this.

```
#ifdef __cplusplus
extern "C" {
#endif

<... your C definitions ...>

#ifdef __cplusplus
};
#endif
```

The latter solution does not work with all C++ compilers; it was tested on GNU C++ compiler only.

[11] The new naming rules are used by default; the naming rules can be changed using the compiler command line switch -N.

[12] The enum hack option has become obsolete with the new naming rules.

[13] C language functions cannot be called directly from TTCN–3; you need at least a wrapper function for them.

Chapter 8. References

- [1] Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 1: Core Language European Telecommunications Standards Institute ES 201 873-1 Version 4.5.1, April 2013
- [2] Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 4: TTCN–3 Operational Semantics European Telecommunications Standards Institute. ES 201 873-4 Version 4.1.1, June 2009
- [3] Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 7: Using ASN.1 with TTCN–3 European Telecommunications Standards Institute. ES 201 873-7 Version 4.5.1, April 2013
- [4] Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 9: Using XML Schema with TTCN–3 European Telecommunications Standards Institute. ES 201 873-9 Version 4.5.1, April 2013
- [5] ITU-T, X.690, Information TechnologyASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)International Telecommunication Union, November 2008
- [6] ITU-T, X.693, Information TechnologyASN.1 encoding rules: XML Encoding Rules (XER), November 2008
- [7] ITU-T, X.693 amendment 1, Information Technology ASN.1 encoding rules: XER encoding instructions and EXTENDED-XER, October 2003
- [8] ISO/IEC 10646-1, Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane, Second edition, 200009-15
- [9] RFC3629: UTF-8, a transformation format of ISO 10646
- [10] User Guide for TITAN TTCN-3 Test Executor
- [11] Installation guide for TITAN TTCN-3 Test Executor
- [12] Release Notes for TITAN TTCN-3 Test Executor
- [13] Technical Reference for TITAN TTCN-3 Test Executor
- [14] David A. Wheeler, Program Library HOWTO
- [15] ETSI ES 202 781 V1.4.1. (2015-06 Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Configuration and Deployment Support)
- [16] ETSI ES 202 782 V1.3.1. (2015-06 Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: TTCN-3 Performance and Real Time Testing)

Chapter 9. Abbreviations

API

Application Programming Interface

ASN.1

Abstract Syntax Notation One

ATS

Abstract Test Suite

BER

Basic Encoding Rules (of ASN.1)

BXER

Basic XER

BNF

Backus–Naur Formalism

CER

Canonical Encoding Rules (of ASN.1)

CXER

Canonical XER

DER

Distinguished Encoding Rules (of ASN.1)

ETS

Executable Test Suite

ETSI

European Telecommunications Standards Institute

EXER

Extended XER

GUI

Graphical User Interface

HC

Host Controller

HTML

Hypertext Markup Language

HTTP
: HyperText Transfer Protocol

IP
: Internet Protocol

LSB
: Least Significant Byte

MC
: Main Controller

MTC
: Main (or Master) Test Component

PDU
: Protocol Data Unit

pl
: Patch Level

PTC
: Parallel Test Component

PT
: Port Type

SO
: Shared Object

SUT
: System Under Test

TC
: Test Component (either MTC or PTC)

TCC
: Test Competence Center

TCP
: Transmission Control Protocol

TLV
: Tag, Length, Value

TTCN
: Tree and Tabular Combined Notation

TTCN–2

Tree and Tabular Combined Notation

TTCN–3

Tree and Tabular Combined Notation version 3 (formerly)
Testing and Test Control Notation (new resolution)

URL

Universal Resource Locator

XER

XML Encoding Rules for ASN.1

XML

Extensible Markup Language