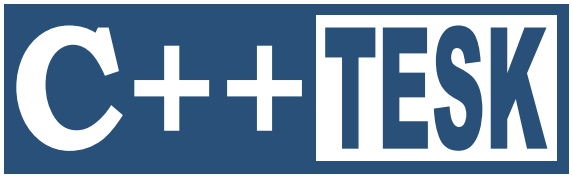

C++TESK Hardware Extension: **Quick Reference**

Version 1.2, 04/09/2013

C++TESK Hardware Extension tool is included into C++TESK Testing ToolKit which can be downloaded from the page http://forge.ispras.ru/projects/cpptesk-toolkit.

Please let us know about your proposals and problems while using C++TESK Testing ToolKit sending them to cpptesk-support@ispras.ru. The forum http://hw-forum.ispras.ru can be also used for such a purpose.

Content

Introduction

This document is a quick reference of C++TESK Hardware Extension tool (C++TESK, hereafter) included into C++TESK Testing ToolKit and aimed to automated development of test systems for HDL-models[1] of hardware. The tool architecture bases on general UniTESK[2] conventions.

Test system is meant to be a special program which evaluates the functional correctness of *target system*, applying some input data (*stimuli*) and analyzing received results of their application (*reactions*). Typical test system consists of three common components: (1) *stimulus generator* making sequences of stimuli (*test sequences*), (2) *test oracle* checking correctness of reactions, and (3) *test completeness evaluator*.

The document describes facilities of C++TESK aimed to development of mentioned test system components, and consists of four common chapters.

- *Development of reference model*
- *Development of reference model adapter*
- *Description of test coverage*
- *Development of test scenario*

First two chapters touch upon development of test oracle basic parts: *reference model* covering functionality of target system, and *reference model adapter* binding reference model with target system. The third chapter is devoted to the test completeness estimation on the base of *test coverage*. The last chapter concerns development of *test scenario* which is the main part of stimulus generator.

More detailed toolkit review is given in «*C++TESK Testing ToolKit: User Guide*».

Installation process of C++TESK is described in «*C++TESK Testing ToolKit: Installation Guide*».

Supporting obsolete constructions

Updates of C++TESK do not interfere in the compilation and running of test systems developed for older versions of C++TESK. When the toolkit has incompatible with older versions update, this update is marked by a build number in form of `yymmdd`, i.e. `110415` – the 15th of April, 2011. The update is available only if macro `CPPTESKHW_VERSION`[3] is appropriately defined. E.g., `-DCPPTESKHW_VERSION=110415` enables usage of incompatible updated having been made by the 15th of April, 2011 (including this date). Each build of the toolkit has the whole list of such changes.

Obsolete constructions of the toolkit are not described in this document. Possibilities of the toolkit, being incompatible with having become obsolete constructions, are presented with the build number, since which they have been available. E.g., the sentence *"the means are supported from 110415 build"* means that usage of these means requires two conditions: (1) using toolkit build has the number 110415 or greater, (2) the compilation option `-DCPPTESKHW_VERSION=`*`number`*, where *`number`* is not less than 110415, is used.

If some obsolete constructions are not used or prevent the toolkit from further development, they might become unsupported. In this case, during compilation of test system using these constructions, a message with advices about correction of test system will be shown.

---

1 HDL (Hardware Description Language) — class of program languages used for description of hardware.

2 http://www.unitesk.ru

3 Using gcc compiler it can be done by the option `-D`*`macro_name=value`*.

During compilation of test systems, developed by means of C++TESK, usage of the compiler option `-DCPPTESKHW_VERSION=number` is **highly recommended**.

Naming convention

The core of the toolkit is developed as a C++ library. Available means are grouped into the following namespaces.

- `cpptesk::hw` — means for development of reference models and their adapters;
- `cpptesk::ts` — basic means for test system development;
- `cpptesk::ts::coverage` — means for test coverage description;
- `cpptesk::ts::engine` — library with test engines[4];
- `cpptesk::tracer` — means for tracing;
- `cpptesk::tracer::coverage` — means for coverage tracing.

A lot of C++TESK means are implemented in form of macros. To avoid conflicts of names, names of all macros start with prefix `CPPTESK_`, e.g., `CPPTESK_MODEL(name)`. Generally, each macro has two aliases: *shortened name* (without `CPPTESK_` prefix) and *short name*, (after additional "compression" of shortened name). E.g., macro `CPPTESK_ITERATION_BEGIN` has two aliases: `ITERATION_BEGIN` and `IBEGIN`. To use shortened and short names, macros `CPPTESK_SHORT_NAMES` and `CPPTESK_SHORT_SHORT_NAMES` should be defined, respectively.

Development of reference model

Reference model is a structured set of classes describing functionality of target system at some abstraction level[5]. Reference model consists of *message classes* describing format of input and output data (structures of stimuli and reactions), *main class* and set of auxiliary classes. Hereafter, a main class of reference model will be meant under term *reference model*.

Class of message

Message classes are declared by macro `CPPTESK_MESSAGE(name)`. Row with macro `CPPTESK_SUPPORT_CLONE(name)` defining the message clone method `name* clone()` should be written inside of the each message class declaration.

Example:

```
#include <hw/message.hpp>

CPPTESK_MESSAGE(MyMessage) {
public:
    CPPTESK_SUPPORT_CLONE(MyMessage)
    ...
};
```

Notice: Row `CPPTESK_SUPPORT_CLONE(name)` is obligatory.

[4] Test engines are toolkit library components used for producing of stimulus sequence. Test engines use test scenario (see chapter *"Development of test scenario"*)

[5] The toolkit allows developing of both abstract functional models and detailed models describing target system cycle-accurately.

Input message randomizer

Virtual method `randomize()` (*randomizer* of the message) should be overloaded in each input message class. There are two macros `CPPTESK_{DECLARE|DEFINE}_RANDOMIZER` for this purpose.

Example:

```
CPPTESK_MESSAGE(MyMessage) {
    ...
    CPPTESK_DECLARE_RANDOMIZER();
private:
    int data;
};

CPPTESK_DEFINE_RANDOMIZER(MyMessage) {
    data = CPPTESK_RANDOM(int);
}
```

The following macros can be used for randomization of data fields.

- `CPPTESK_RANDOM(type)` — generation of random integer value;
- `CPPTESK_RANDOM_WIDTH(type, length)` — generation of random integer value of given number of bits;
- `CPPTESK_RANDOM_FIELD(type, min_bit, max_bit)` — generation of random integer value with zero bits outside the given range;
- `CPPTESK_RANDOM_RANGE(type, min_value, max_value)` — generation of random integer value from given integer range;
- `CPPTESK_RANDOM_CHOICE(type, value_1, ..., value_n)` — random choice from given set of any type values.

Notice: randomizer may not be defined if all data fields are defined by special macros (see chapter "*Message data fields*").

Comparator of output messages

Virtual method `compare()` (*comparator* of messages) should be overloaded in each output message class. There are two macro `CPPTESK_{DECLARE|DEFINE}_COMPARATOR` for this purpose (build is not less than 110428).

Example:

```
CPPTESK_MESSAGE(MyMessage) {
    ...
    CPPTESK_DECLARE_COMPARATOR();
private:
    int data;
};

CPPTESK_DEFINE_COMPARATOR(MyMessage) {
    const MyMessage &rhs = CPPTESK_CONST_CAST_MESSAGE(MyMessage);
    // in case of difference between messages return not empty string
    if(data != rhs.data)
        { return "incorrect data"; }
    // empty string is interpreted as absence of difference
    return COMPARE_OK;
}
```

Notice: comparator may not be defined if all data fields are defined by special macros (see chapter "*Message data fields*").

Message data fields

The following macros are aimed to declaration of integer *data fields*.

- `CPPTESK_DECLARE_FIELD(name, length)`
  declares integer field with given name and length.
- `CPPTESK_DECLARE_MASKED_FIELD(name, length, mask)`
  declares integer field with given name, length, and mask.
- `CPPTESK_DECLARE_BIT(name)`
  declares bit field with given name.

Example:

```
#include <hw/message.hpp>

CPPTESK_MESSAGE(MyMessage) {
public:
    CPPTESK_DECLARE_MASKED_FIELD(addr, 32, 0xffffFFF0);
    CPPTESK_DECLARE_FIELD(data, 32);
    CPPTESK_DECLARE_BIT(flag);
    ...
};
```

Notice: length of the data fields should not exceed 64 bits.

All declared data fields should be registered in message class constructor by means of macro `CPPTESK_ADD_FIELD(full_name)` or, when the data field should not be taken into account by comparator, by means of `CPPTESK_ADD_INCOMPARABLE_FIELD(full_name)`.

Example:

```
MyMessage::MyMessage() {
    CPPTESK_ADD_FIELD(MyMessage::addr);
    CPPTESK_ADD_FIELD(MyMessage::data);
    CPPTESK_ADD_INCOMPARABLE_FIELD(MyMessage::flag);
    ...
}
```

Notice: full_name means usage both name of method and name of class.

Optional messages

Output message can be declared to be *optional* (not obligatory for receiving) if the following method is used.

```
void setOptional(optional_or_not_optional);
```

Notice: Default value of the parameter “optional_or_not_optional” is true, i.e. if there has not been correspondent implementation output message by a certain timeout, the message will be simply deleted without showing of an error. At the same time, the optional message is added to the *interface arbiter* and might affect to the matching of other output messages. If correspondent implementation reactions are received after the timeout and they are to be ignored, the flag of the message being optional should be set in message class constructor.

Reference model

Reference model (*main class of the reference model*) is declared by macro `CPPTESK_MODEL(name)`.

Example:

```
#include <hw/model.hpp>
```

```
CPPTESK_MODEL(MyModel) {
    ...
};
```

Reference model contains declaration of *input* and *output interfaces*, *operations*, auxiliary processes, and data necessary for operation description.

Interface

Input and *output interfaces* of the reference model are declared by means of two macros `CPPTESK_DECLARE_{INPUT|OUTPUT}(name)`, respectively.

Example:

```
#include <hw/model.hpp>

CPPTESK_MODEL(MyModel) {
public:
    CPPTESK_DECLARE_INPUT(input_iface);
    CPPTESK_DECLARE_OUTPUT(output_iface);
    ...
};
```

All declared interfaces should be registered in reference model constructor by means of two macros `CPPTESK_ADD_{INPUT|OUTPUT}(name)`.

Example:

```
MyModel::MyModel() {
    CPPTESK_ADD_INPUT(input_iface);
    CPPTESK_ADD_OUTPUT(output_iface);
    ...
}
```

Setting up ignoring of failures on output interface

To disable showing errors of a certain type on a given interface is possible by means of the method:

```
void setFailureIgnoreTypes(disabling_error_types_mask);
```

Disabling error types include errors of implementation reaction absence (`MISSING_REACTION`) and specification reaction absence (`UNEXPECTED_REACTION`) on the given interface. Error types can be grouped by bit operation “or”.

Process

Processes are the main means of functional specification of hardware. Processes are subdivided into *operations* (see chapter “*Operation*”) and *internal processes*. Operations describe processing of stimuli of a certain types by the target system. Internal processes are used for definition of the other, more complex processes, including operations.

Declaration and definition of reference model processes are made by means of macros `CPPTESK_{DECLARE|DEFINE}_PROCESS(name)`. Definition of the process should be started by calling macro `CPPTESK_START_PROCESS()`, and finished by calling macro `CPPTESK_STOP_PROCESS()`.

Example:

```
#include <hw/model.hpp>

CPPTESK_MODEL(MyModel) {
public:
    ...
    CPPTESK_DECLARE_PROCESS(internal_process);
    ...
};

CPPTESK_DEFINE_PROCESS(MyModel::internal_process) {
    CPPTESK_START_PROCESS();
    ...
    CPPTESK_STOP_PROCESS();
}
```

Notice: macro `CPPTESK_START_PROCESS()` may be used only once in definition of the process, and usually precedes the main process code. Semantics of `CPPTESK_STOP_PROCESS()` is similar to the semantics of operator `return` — when the macro is called, the process finished.

Notice: keyword `process` is reserved and cannot be used for naming of processes.

Process parameters

Process should have three obligatory *parameters*: (1) process execution *context*, (2) associated with process *interface*, and (3) *message*. To access process parameters is possible by means of the following macros.

- `CPPTESK_GET_PROCESS()` — get process context;
- `CPPTESK_GET_IFACE()` — get process interface;
- `CPPTESK_GET_MESSAGE()` — get message.

To cast message parameter to the necessary type is possible by means of the following macros.

- `CPPTESK_CAST_MESSAGE(message_class);`
- `CPPTESK_CONST_CAST_MESSAGE(message_class).`

Example:

```
#include <hw/model.hpp>

CPPTESK_DEFINE_PROCESS(MyModel::internal_process) {
    // copy message to the local variable
    MyMessage msg = CPPTESK_CAST_MESSAGE(MyMessage);
    // get reference to the message
    MyMessage &msg_ref = CPPTESK_CAST_MESSAGE(MyMessage);
    // get constant reference to the message
    const MyMessage &const_msg_ref = CPPTESK_CONST_CAST_MESSAGE(MyMessage);
    ...
}
```

Process priority

During execution, each process is assigned with a *priority*, unsigned integer value from range [1, 255] (0 is reserved). Priority affects the order of process execution inside of one cycle (processes with higher priority run first). Priorities may be used in matching of implementation and specification reactions (see chapter "*Reaction arbiter*"). When started, all processes are assigned with the same priority (`NORMAL_PRIORITY`). To change the priority is possible by means of the following macros.

- `CPPTESK_GET_PRIORITY()` — get process priority.
- `CPPTESK_SET_PRIORITY(priority)` — set process priority.

Some priority values are defined in the enumeration type `priority_t` (`cpptesk::hw` namespace). The most general of them are the following ones.

- `NORMAL_PRIORITY` — normal priority;
- `LOWEST_PRIORITY` — the lowest priority;
- `HIGHEST_PRIORITY` — the highest priority.

Example:

```
#include <hw/model.hpp>
#include <iostream>

CPPTESK_DEFINE_PROCESS(MyModel::internal_process) {
    ...
    std::cout << "process priority is " << std::dec
              << CPPTESK_GET_PRIORITY() << std::end;
    ...
    CPPTESK_SET_PRIORITY(cpptesk::hw::HIGHEST_PRIORITY);
    ...
}
```

Modeling of delays

To model *delays* in processes is possible by means of the following macros.

- `CPPTESK_CYCLE()`
  delay of one cycle.
- `CPPTESK_DELAY(number_of_cycles)`
  delay of several cycles.
- `CPPTESK_WAIT(condition)`
  delay till condition is satisfied.
- `CPPTESK_WAIT_TIMEOUT(condition, timeout)`
  limited in time delay till condition is satisfied.

Example:

```
#include <hw/model.hpp>
#include <iostream>

CPPTESK_DEFINE_PROCESS(MyModel::internal_process) {
    ...
    std::cout << "cycle: " << std::dec << time() << std::end;
    // delay of one cycle
    CPPTESK_CYCLE();
    std::cout << "cycle: " << std::dec << time() << std::end;
    ...
    // wait till outputs.ready is true,
    // but not more than 100 cycles
    CPPTESK_WAIT_TIMEOUT(outputs.ready, 100);
    ...
}
```

Process calling

Reference model *process calling* from another process is made by means of macro `CPPTESK_CALL_PROCESS(mode, process_name, interface, message)`, where

mode might be either PARALLEL or SEQUENTIAL. In the first case separated process is created, which is executed in parallel with the parent process. In the second case consequent execution is performed, where returning to the parent process execution is possible only after child process has been finished.

Example:

```
#include <hw/model.hpp>

CPPTESK_DEFINE_PROCESS(MyModel::some_process) {
    ...
    // call separated process
    CPPTESK_CALL_PROCESS(PARALLEL, internal_process,
        CPPTESK_GET_IFACE(), CPPTESK_GET_MESSAGE());
    ...
    // call process and wait till it has been finished
    CPPTESK_CALL_PROCESS(SEQUENTIAL, internal_process,
        CPPTESK_GET_IFACE(), CPPTESK_GET_MESSAGE());
    ...
}
```

Notice: to call *new* processes is possible from any reference model methods, not only from methods describing processes. Macro CPPTESK_CALL_PARALLEL(*process_name*, *interface*, *message*) should be used in this case. Calling process with mode SEQUENTIAL from the method not being a process is prohibited.

Stimulus receiving

Inside of the process, *receiving of stimulus* on one of the input interfaces can be modeled. It is possible by means of macro CPPTESK_RECV_STIMULUS(*mode*, *interface*, *message*). Executing this macro, test system applies the stimulus to the target system via adapter of the correspondent input interface (see chapter «*Input interface adapter*»). Semantics of the *mode* parameter is described in the chapter «*Process calling*».

Example:

```
#include <hw/model.hpp>

CPPTESK_DEFINE_PROCESS(MyModel::some_process) {
    // modeling of stimulus receiving
    CPPTESK_RECV_STIMULUS(PARALLEL, input_iface, input_msg);
    ...
}
```

Reaction sending

Modeling of *reaction sending* is done by the macro CPPTESK_SEND_REACTION(*mode*, *interface*, *message*). Executing this macro, test system calls adapter of the correspondent output interface. The adapter starts waiting for the proper implementation reaction. When being received, the reaction is transformed into object of the correspondent message class (see chapter "*Adapter of the output interface*"). Then test system compares reference message with received message by means of comparator (see chapter "*Comparator of output messages*"). Semantics of the *mode* parameter is described in the chapter "*Process calling*".

Example:

```
#include <hw/model.hpp>

CPPTESK_DEFINE_PROCESS(MyModel::some_process) {
    // modeling of reaction sending
    CPPTESK_SEND_REACTION(SEQUENTIAL, output_iface, output_msg);
    ...
}
```

Operation

Declaration and definition of *interface operations* of reference model is made by means of macros `CPPTESK_{DECLARE|DEFINE}_STIMULUS(name)`. The definition should start from macro `CPPTESK_START_STIMULUS(mode)`, and stop by macro `CPPTESK_STOP_STIMULUS()`.

Example:

```
#include <hw/model.hpp>

CPPTESK_MODEL(MyModel) {
public:
    CPPTESK_DECLARE_STIMULUS(operation);
    ...
};

CPPTESK_DEFINE_STIMULUS(MyModel::operation) {
    CPPTESK_START_STIMULUS(PARALLEL);
    ...
    CPPTESK_STOP_STIMULUS();
}
```

Notice: operations are particular cases of processes, so that all the constructions from chapter "*Process*" can be used in them.

Notice: calling macro `CPPTESK_START_STIMULUS(mode)` is equivalent to the calling macro `CPPTESK_RECV_STIMULUS(mode, ...)`, where interface and message parameters are assigned with correspondent operation parameters.

Callback function

In the main class of reference model, several callback functions are defined. The functions can be overloaded in the reference model. The main callback function is `onEveryCycle()`.

Function onEveryCycle

Function `onEveryCycle()` is called at the beginning of each reference model execution cycle.

Example:

```
#include <hw/model.hpp>

CPPTESK_MODEL(MyModel) {
public:
    virtual void onEveryCycle();
    ...
};

void MyModel::onEveryCycle() {
    std::cout << "onEveryCycle: time=" << std::dec << time() << std::endl;
}
```

Development of reference model adapter

Reference model adapter (*mediator*) is a component of test system, binding reference model with target system. The adapter serializes input message objects into sequences of input signal values, deserializes sequences of output signal values into output message objects, and matches received from target system reactions with reference values.

Reference model adapter

Reference model adapter is a subclass of reference model class. It is declared by means of the macro `CPPTESK_ADAPTER(adapter_name, model_name)`.

Example:

```
#include <hw/media.hpp>

CPPTESK_ADAPTER(MyAdapter, MyModel) {
    ...
};
```

Synchronization methods, *input* and *output interface adapters*, *output interface listeners*, and *reaction arbiters* are declared in reference model adapter.

Synchronizer

Synchronizer is a low-level part of reference model adapter, responsible for synchronization of test system with being tested HDL-model. Synchronizer is implemented by overloading the following five methods of reference model adapter.

- `void initialize()` — test system initialization;
- `void finialize()` — test system finalization;
- `void setInputs()` — synchronization of inputs;
- `void getOutputs()` — synchronization of outputs;
- `void simulate()` — synchronization of time.

When hardware models written in Verilog being verified, these methods can be implemented by standard interface VPI (Verilog Procedural Interface). Also, tool VeriTool[6] can be used for automation of synchronizer development. In this case, macro `CPPTESK_VERITOOL_ADAPTER(`*`adapter_name`*`, `*`model_name`*`)` can be used for facilitating of the efforts.

Example:

```
#include <hw/veritool/media.hpp>
// file generated by tool VeriTool
#include <interface.h>

CPPTESK_VERITOOL_ADAPTER(MyAdapter, MyModel) {
    ...
};
```

Used for definition of synchronization methods functions and data structures (fields `inputs` and `outputs`) are generated automatically by tool VeriTool analyzing Verilog hardware model interface.

Notice: when macro `CPPTESK_VERITOOL_ADAPTER` being used, fields `inputs` и `outputs` **should not** be declared and methods and methods of synchronizer **should not** be overloaded.

Notice: tool VeriTool provides access to values of all HDL-model signals, including internal ones. To get access is possible by means of macros `CPPTESK_GET_SIGNAL(`*`signal_type`*`, `*`signal_name`*`)` for getting value of signal testbench.target.signal_name (*signal_type* is meant to be from the following list: int, uint64_t, etc.), and `CPPTESK_SET_SIGNAL(`*`signal_type`*`, `*`signal_name`*`, `*`new_value`*`)` for setting to a new value the signal testbench.target.signal_name (*signal_type* is meant to be the same as for getting value macro).

[6] http://forge.ispras.ru/projects/veritool

Input interface adapter

Input interface adapter is a process defined in reference model adapter and bound with one of the input interfaces. Input interface adapter is called by `CPPTESK_START_STIMULUS(mode)` macro (see chapter "*Operation*") or by `CPPTESK_RECV_STIMULUS(mode, interface, message)` macro (see chapter "*Stimulus receiving*"). Declaration and definition of input interface adapters are done in typical for processes way.

It should be noticed, that just before the serialization, the input interface adapter should *capture* the interface. Capturing is made by macro `CPPTESK_CAPTURE_IFACE()`. Correspondently, after the serialization, the interface should be released by macro `CPPTESK_RELEASE_IFACE()`.

Example:

```
#include <hw/media.hpp>

CPPTESK_ADAPTER(MyAdapter, MyModel) {
    CPPTESK_DECLARE_PROCESS(serialize_input);
    ...
};

CPPTESK_DEFINE_PROCESS(MyAdapter::serialize_input) {
    MyMessage msg = CPPTESK_CAST_MESSAGE(MyMessage);
    // start serialization process
    CPPTESK_START_PROCESS();
    // capture input interface
    CPPTESK_CAPTURE_IFACE();
    // set operation start strobe
    inputs.start = 1;
    // set information signals
    inputs.addr  = msg.get_addr();
    inputs.data  = msg.get_data();
    // one cycle delay
    CPPTESK_CYCLE();
    // reset of operation strobe
    inputs.start = 0;
    // release input interface
    CPPTESK_RELEASE_IFACE();
    // stop serialization process
    CPPTESK_STOP_PROCESS();
}
```

Binding of adapter and interface is made in reference model constructor by means of macro `CPPTESK_SET_INPUT_ADAPTER(interface_name, adapter_full_name)`.

Example:

```
MyAdapter::MyAdapter{
    CPPTESK_SET_INPUT_ADAPTER(input_iface, MyAdater::serialize_input);
    ...
};
```

Notice: when input interface adapter being registered, its full name (including the name of reference model adapter class) should be used.

Output interface adapter

Output interface adapter is a process defined in reference model adapter and bound with one of the output interfaces, Output interface adapter is called by `CPPTESK_SEND_REACTION(mode, interface, message)` macro (see chapter "*Reaction sending*"). Declaration and definition of output interface adapters are done by means of the following macros.

- `CPPTESK_WAIT_REACTION(condition)`
  wait for reaction and allow reaction arbiter to access the reaction;
- `CPPTESK_NEXT_REACTION()`
  releasing of reaction arbiter (see chapter "*Reaction arbiter*").

Example:

```
#include <hw/media.hpp>

CPPTESK_ADAPTER(MyAdapter, MyModel) {
    CPPTESK_DECLARE_PROCESS(deserialize_output);
    ...
};

CPPTESK_DEFINE_PROCESS(MyAdapter::deserialize_input) {
    // get reference to the message object
    MyMessage &msg = CPPTESK_CAST_MESSAGE(MyMessage);
    // start deserialization process
    CPPTESK_START_PROCESS();
    // wait for result strobe
    CPPTESK_WAIT_REACTION(outputs.result);
    // read data
    msg.set_data(outputs.data);
    // release reaction arbiter
    CPPTESK_NEXT_REACTION();
    // stop deserialization process
    CPPTESK_STOP_PROCESS();
}
```

The waiting for the implementation reaction time is restricted by a *timeout*. The timeout is set by macro `CPPTESK_SET_REACTION_TIMEOUT(timeout)`, which as well as macro `CPPTESK_SET_OUTPUT_ADAPTER(interface_name, adapter_full_name)` is called in constructor of reference model adapter.

Example:

```
MyAdapter::MyAdapter{
    CPPTESK_SET_OUTPUT_ADAPTER(output_iface, MyAdater::deserialize_output);
    CPPTESK_SET_REACTION_TIMEOUT(100);
    ...
};
```

Notice: when output interface adapter being registered, its full name (including the name of reference model adapter class) should be used.

Output interface listener (*deprecated feature*)

Output interface listener is a special-purpose process, waiting for appearing of implementation reactions at the beginning of each cycle, and registering error in case of unexpected reactions. Definition of listeners is made by `CPPTESK_DEFINE_BASIC_OUTPUT_LISTENER(name, interface_name, condition)` macro.

Example:

```
#include <hw/media.hpp>

CPPTESK_ADAPTER(MyAdapter, MyModel) {
    CPPTESK_DEFINE_BASIC_OUTPUT_LISTENER(output_listener,
        output_iface, outputs.result);
    ...
};
```

Output interface listener is started in constructor of reference model adapter by macro `CPPTESK_CALL_OUTPUT_LISTENER(listener_full_name, interface_name)`.

Example:

```
MyAdapter::MyAdapter{
    ...
    CPPTESK_CALL_OUTPUT_LISTENER(MyAdapter::output_listener, output_iface);
    ...
};
```

Reaction arbiter

Reaction arbiter (*output interface arbiter*) is aimed for matching *implementation reactions* (received from HDL-model) with *specification reactions* (calculated by reference model). Having been matched, the reaction pairs are sent to comparator, showing an error if there is difference in data between two reactions (see chapter "*Comparator of output messages*").

The common types of arbiters are the following.

- `CPPTESK_FIFO_ARBITER` — implementation reaction having been received, the arbiter prefers specification reaction which was created by the earliest among the other reactions call of macro `CPPTESK_SEND_REACTION()` (see chapter "*Reaction sending*").
- `CPPTESK_PRIORITY_ARBITER` — the arbiter prefers specification reaction which was created with the highest priority by macro `CPPTESK_SEND_REACTION()` (see chapter "*Process priority*").

To declare reaction arbiter in reference model adapter class is possible by means of macro `CPPTESK_DECLARE_ARBITER(`*`type`*`, `*`name`*`)`. To bind arbiter with output interface is possible by means of macro `CPPTESK_SET_ARBITER(`*`interface`*`, `*`arbiter`*`)`, which should be called in constructor of reference model adapter.

Example:

```
#include <hw/media.hpp>

CPPTESK_ADAPTER(MyAdapter, MyModel) {
    CPPTESK_DECLARE_ARBITER(CPPTESK_FIFO_ARBITER, output_iface_arbiter);
    ...
};

MyAdapter::MyAdapter() {
    CPPTESK_SET_ARBITER(output_iface, output_iface_arbiter);
    ...
}
```

Test coverage description

Test coverage is aimed for evaluation of test completeness. As a rule, test coverage structure is described explicitly by enumerating of all possible in the test situations (*test situations*). To describe complex test situations, composition of simpler test coverage structures is used.

Test coverage can be described in the main class of reference model or moved to external class (*test coverage class*). In the second case, the class with test coverage description should have a reference to the reference model (see chapter "*Test coverage class*").

Class of test coverage

Class of test coverage is a class containing *definition of test coverage structure* and *functions calculating test situations*. As test coverage is defined in terms of reference model, test coverage class should have a reference to the main class of reference model. To trace test situations, test coverage class has *test situation tracer* — an object of `CoverageTracker` class (namespace `cpptesk::tracer::coverage`) and *function tracing test situations*.

Example:

```
#include <ts/coverage.hpp>
#include <tracer/tracer.hpp>

class MyModel;

// Declaration of test coverage class
class MyCoverage {
public:
    MyCoverage(MyModel &model): model(model) {}

    // Test situation tracer
    CoverageTracker tracker;

    // Description of test coverage structure
    CPPTESK_DEFINE_ENUMERATED_COVERAGE(MY_COVERAGE, "My coverage", (
         (SITUATION_1, "Situation 1"),
         ...
         (SITUATION_N, "Situation N")
    ));

    // Function calculating test situation: signature of the function
    // contains all necessary for it parameters
    MY_COVERAGE cov_MY_COVERAGE(...) const;

    // Function tracing test situations: signature of the function
    // is the same as signature of the previous function
    void trace_MY_COVERAGE(...);
    ...
private:
    // Reference to the reference model
    MyModel &model;
};
```

Test coverage structure

Test coverage structure is described by means of *enumerated coverage*, *coverage compositions*, *excluded coverage composition*, and *test coverage aliases*.

Enumerated coverage

Enumerated coverage, as it goes from the coverage name, is defined by explicit enumeration of all possible test situations by macro `CPPTESK_DEFINE_ENUMERATED_COVERAGE(`*`coverage`*`, `*`description`*`, `*`situations`*`)`, where *`coverage`* is an identifier of the coverage type, *`description`* is a string, and *`situations`* is the list of situations like `((`*`id`*`, `*`description`*`), ...)`.

Example:

```
#include <ts/coverage.hpp>

CPPTESK_DEFINE_ENUMERATED_COVERAGE(FIFO_FULLNESS, "FIFO fullness", (
    (FIFO_EMPTY, "Empty"),
     ...
    (FIFO_FULL, "Full")
));
```

Coverage composition

Coverage composition allows creation of test situation structure basing on two test coverage structures, containing Cartesian product of situations from both initial structures. Coverage composition is made by means of macro `CPPTESK_DEFINE_COMPOSED_COVERAGE(`*`type`*`,`

description, *coverage_1*, *coverage_2*). Description of new test situations is made according to the pattern "%s,%s".

Example:

```
#include <ts/coverage.hpp>

CPPTESK_DEFINE_ENUMERATED_COVERAGE(COVERAGE_A, "Coverage A", (
    (A1, "A one"),
    (A2, "A two")
));

CPPTESK_DEFINE_ENUMERATED_COVERAGE(COVERAGE_B, "Coverage B", (
    (B1, "B one"),
    (B2, "B two")
));

// Product of structures A and B makes the following situations:
// (COVERAGE_AxB::Id(A1, B1), "A one, B one")
// (COVERAGE_AxB::Id(A1, B2), "A one, B two")
// (COVERAGE_AxB::Id(A2, B1), "A two, B one")
// (COVERAGE_AxB::Id(A2, B2), "A two, B two")
CPPTESK_DEFINE_COMPOSED_COVERAGE(COVERAGE_AxB, "Coverage AxB",
    COVERAGE_A, COVERAGE_B);
```

Excluded coverage composition

To make product of test coverage structures and exclude *unreachable* test situations is possible by macro CPPTESK_DEFINE_COMPOSED_COVERAGE_EXCLUDING(*type*, *description*, *coverage_1*, *coverage_2*, *excluded*), where *excluded* is the list like ({*coverage_1*::*id*, *coverage_2*::*id*}, ...). Instead of test situation identifier, macro ANY() can be used. To product coverage structures being products themselves, correspondent tuples should be used instead of pairs.

Example:

```
#include <ts/coverage.hpp>

CPPTESK_DEFINE_ENUMERATED_COVERAGE(COVERAGE_A, "Coverage A", (
    (A1, "A one"),
    (A2, "A two")
));

CPPTESK_DEFINE_ENUMERATED_COVERAGE(COVERAGE_B, "Coverage B", (
    (B1, "B one"),
    (B2, "B two")
));

// The following composition makes the following test situations:
// (COVERAGE_AxB::Id(A1, B2), "A one, B two")
// (COVERAGE_AxB::Id(A2, B1), "A two, B one")
// (COVERAGE_AxB::Id(A2, B2), "A two, B two")
CPPTESK_DEFINE_COMPOSED_COVERAGE_EXCLUDING(COVERAGE_AxB, "Coverage AxB",
    COVERAGE_A, COVERAGE_B, ({COVERAGE_A::A1, COVERAGE_B::B1}));
```

Test coverage alias

To make a test coverage alias (test coverage with different name, but with the same test situations), macro CPPTESK_DEFINE_ALIAS_COVERAGE(*alias*, *description*, *coverage*) should be used.

Example:

```
#include <ts/coverage.hpp>
```

```
CPPTESK_DEFINE_ENUMERATED_COVERAGE(COVERAGE_A, "Coverage A", (
    (A1, "A one"),
    (A2, "A two")
));

// COVERAGE_B – alias of COVERAGE_A
CPPTESK_DEFINE_ALIAS_COVERAGE(COVERAGE_B, "Coverage B", COVERAGE_A);
```

Calculating current test situation function

Calculating current test situation function is a function, which returns identifier of the current test situation (see chapter "*Structure of the test coverage*"), having analyzed the reference model state and (possibly) input parameters of the operation. Identifier of test situation for enumerated coverage looks like *`coverage::identifier`* and *`class::coverage::identifier`* when used outside of test coverage class. Calculating current test situation function for production of coverage structures can be obtained by calling functions for particular coverage structures and "production" of their results (operator * should be appropriately overloaded).

Example:

```
#include <ts/coverage.hpp>

class MyCoverage {
    // definition of the enumerated coverage structure COVERAGE_A
    CPPTESK_DEFINE_ENUMERATED_COVERAGE(COVERAGE_A, "Coverage A", (
        (A1, "A one"),
        (A2, "A two")
    ));
    // test situation calculating function for coverage COVERAGE_A
    COVERAGE_A cov_COVERAGE_A(int a) const {
        switch(a) {
        case 1: return COVERAGE_A::A1;
        case 2: return COVERAGE_A::A2;
        }
        assert(false);
    }

    // definition of COVERAGE_B – alias of COVERAGE_A
    CPPTESK_DEFINE_ALIAS_COVERAGE(COVERAGE_B, "Coverage B", COVERAGE_A);
    // test situation calculating function for coverage COVERAGE_B
    COVERAGE_B cov_COVERAGE_B(int b) const {
        return cov_COVERAGE_A(b);
    }

    // definition of COVERAGE_AxB – production of COVERAGE_A and COVERAGE_B
    CPPTESK_DEFINE_COMPOSED_COVERAGE(COVERAGE_AxB, "Coverage AxB",
        COVERAGE_A, COVERAGE_B);s
    // test situation calculating function of coverage COVERAGE_AxB
    COVERAGE_AxB cov_COVERAGE_AxB(int a, int b) const {
        return cov_COVERAGE_A(a) * cov_COVERAGE_B(b);
    }
    ...
};
```

Tracing test situation function

Tracing test situation function is defined for each upper-level test coverage structure. As tracing function calls test situation calculating function, their parameters usually coincide. Implementation of this function is based on test situation tracer, being an object of class `CoverageTracker` (namespace `cpptesk::tracer::coverage`).

Example:

```
#include <ts/coverage.hpp>
#include <tracer/tracer.hpp>

CoverageTracker tracer;

void trace_COVERAGE_A(int a) {
    tracer << cov_COVERAGE_A(a);
}
```

Development of test scenario

Test scenario is a high-level specification of test, which being interpreted by *test engine* (see chapter "*Test scenario running*") is used by test system for test sequence generation. Test scenario is developed as a special class named *scenario class*.

Class of scenario

Scenario class is declared by macro `CPPTESK_SCENARIO(name)`.

Example:

```
#include <ts/scenario.hpp>

CPPTESK_SCENARIO(MyScenario) {
    ...
private:
    // testing is done via reference model adapter
    MyAdapter dut;
};
```

Test scenario initialization and *finalization methods*, *scenario methods*, and *current state function* are declared in scenario class.

Test scenario initialization method

Test scenario initialization method includes actions which should have been made right before test start. It is defined by overloading of base class virtual method `bool init(int argc, char **argv)`. It returns `true` in case of successful initialization and `false` in case of some problem.

Exmple:

```
#include <ts/scenario.hpp>

CPPTESK_SCENARIO(MyScenario) {
public:
    virtual bool init(int argc, char **argv) {
        dut.initialize();
        std::cout << "Test has started..." << std::endl;
    }
    ...
};
```

Test scenario finalizing method

Test scenario finalizing method contains actions which should be done right after test finish. It is defined by overloading of base class virtual method `void finish()`.

Example:

```
#include <ts/scenario.hpp>

CPPTESK_SCENARIO(MyScenario) {
public:
    virtual void finish() {
        dut.finalize();
        std::cout << "Test has finished..." << std::endl;
```

```
    }
    ...
};
```

Scenario method

Scenario methods iterate parameters of input messages and run operations by means of reference model adapter. One scenario class may contain several scenario method declarations. Scenario method returns value of `bool` type, which is interpreted as a flag of some problem. The only parameter of scenario method is an *iteration context*, which is an object containing variables to be iterated by scenario method (*iteration variables*). Definition of scenario method starts from calling macro `CPPTESK_ITERATION_BEGIN`, and finishes with `CPPTESK_ITERATION_END`.

Example:

```
#include <ts/scenario.hpp>

CPPTESK_SCENARIO(MyScenario) {
public:
    bool scenario(cpptesk::ts::IntCtx &ctx);
    ...
};

bool MyScenario::scenario(cpptesk::ts::IntCtx &ctx) {
    CPPTESK_ITERATION_BEGIN
    ...
    CPPTESK_ITERATION_END
}
```

Scenario methods are registered by macro `CPPTESK_ADD_SCENARIO_METHOD(`*`full_name`*`)` in scenario class constructor.

Example:

```
MyScenario::MyScenario() {
    CPPTESK_ADD_SCENARIO_METHOD(MyScenario::scenario);
    ...
}
```

Access to iteration variables

Iteration variables are fields of iteration context, which is a parameter of scenario method. To access iteration variables is possible by macro `CPPTESK_ITERATION_VARIABLE(`*`name`*`)`, where *`name`* is a name of one of the iteration context fields.

Example:

```
#include <ts/scenario.hpp>

bool MyScenario::scenario(cpptesk::ts::IntCtx &ctx) {
    // get reference to iteration variable
    int &i = CPPTESK_ITERATION_VARIABLE(i);
    CPPTESK_ITERATION_BEGIN
    for(i = 0; i < 10; i++) {
        ...
    }
    CPPTESK_ITERATION_END
}
```

Test action block

Test action (preparation of input message and start of operation) is made in a code block `CPPTESK_ITERATION_ACTION{ ... }` located in scenario method.

Example:

```
#include <ts/scenario.hpp>

...
CPPTESK_ITERATION_BEGIN
for(i = 0; i < 10; i++) {
     ...
     // test action block
     CPPTESK_ITERATION_ACTION {
         // input message randomization
         CPPTESK_RANDOMIZE_MESSAGE(input_msg);
         input_msg.set_addr(i);
         // start operation
         CPPTESK_CALL_STIMULUS_OF(dut, MyModel::operation,
             dut.input_iface, input_msg);
         ...
     }
}
CPPTESK_ITERATION_END
```

Scenario action finishing

Each iteration of scenario method is finished by `CPPTESK_ITERATION_YIELD(`*`verdict`*`)` macro, quitting from scenario method. When being called next time, scenario method will continue its execution from next iteration.

Example:

```
#include <ts/scenario.hpp>

...
CPPTESK_ITERATION_BEGIN
for(i = 0; i < 10; i++) {
     ...
     // test action block
     CPPTESK_ITERATION_ACTION {
         ...
         // quit from scenario method and return verdict
         CPPTESK_ITERATION_YIELD(dut.verdict());
     }
}
CPPTESK_ITERATION_END
```

Delays

Making *delays* (sending of stimuli at different time of HDL-model simulation) in tests requires development of at least one method with calling reference model method `cycle()`. In case of possibility of parallel stimulus running, the most convenient way of usage method `cycle()` is to call this method from purposely created scenario method `nop()`[7]. Notice that in this case method `cycle()` should not be called from any other method.

Пример:

```
#include <ts/scenario.hpp>

bool MyScenario::nop(cpptesk::ts::IntCtx& ctx) {
    CPPTESK_ITERATION_BEGIN
    CPPTESK_ITERATION_ACTION {
        dut.cycle();
        CPPTESK_ITERATION_YIELD(dut.verdict());
    }
    CPPTESK_ITERATION_END
}
```

[7] Name of this method is unrestricted.

Notice: scenario method `nop()` should be registered before any other scenario methods.

Calculating current state function

Calculating current state function is needed for test engines, using exploration of target system state graph for creation of test sequences. Returning by function value is interpreted as system state. Type of the returning value and function name are unrestricted. Method does not allow parameters.

Example:

```
#include <ts/scenario.hpp>

CPPTESK_SCENARIO(MyScenario) {
public:
    ...
    int get_model_state() {
        return dut.buffer.size();
    }
};
```

Setting up of calculating current state function is made by method `void setup(...)`. in test scenario constructor.

Example:

```
#include <ts/scenario.hpp>

MyScenario::MyScenario() {
    setup("My scenario",
    UseVirtual::init,
    UseVirtual::finish,
    &MyScenario::get_model_state);
    ...
}
```

Test scenario running

Test scenario running at local computer is made by calling function `localmain(`*`engine`*`, `*`scenario`*`.getTestScenario(), `*`argc`*`, `*`argv`*`)` (namespace `cpptesk::ts`).

Available test engines are the following (namespace `cpptesk::ts::engine`).

- `fsm` — generator of test sequence based on state graph exploration;
- `rnd` — generator of randomized test sequence.

Example:

```
#include <netfsm/engines.hpp>

using namespace cpptesk::ts;
using namespace cpptesk::ts::engine;
...
MyScenario scenario;
localmain(fsm, scenario.getTestScenario(), argc, argv);
```

Auxiliary possibilities

C++TESK toolkit includes the following auxiliary possibilities: *assertions* and *debug print*. These possibilities can be used in reference models and in all test system components (adapters, test scenarios, etc). Their main aim is to facilitate *debug* of test system.

Assertions

Assertions are predicates (logic constructions) used for description of program properties and, as a rule, checked during runtime. If assertion is violated (predicate shows false), error is fixed and program is stopped. To make assertions is possible by `CPPTESK_ASSERTION(predicate, description)` macro, where *predicate* is a checking property, and *description* is a string describing error bound with violation of this property.

Example:

```
#include <hw/assertion.hpp>

...
CPPTESK_ASSERTION(pointer, "pointer is null");
```

Debug print

Debug print is devoted to debug of test system. In contrast to typical printing by means of, e.g., STL streams, adjustment of debug print is easier (turning on/off, changing of printing color, etc).

Debug print macros

Debug print is commonly made by macro `CPPTESK_DEBUG_PRINT(level, message)`, where *level* is a level of debug print (see chapter *"Debug print levels"*) and *message* is a printing *debug message*, and macro `CPPTESK_DEBUG_PRINTF(level, formal, parameters)`, where `(format, parameters)` is a formatted string and values of used in the string parameters in the same format as they are used by C library function `printf()`.

Example:

```
#include <hw/debug.hpp>

using namespace cpptesk::hw;
...
CPPTESK_DEBUG_PRINT(DEBUG_USER, "The input message is "
    << CPPTESK_GET_MESSAGE());
...
CPPTESK_DEBUG_PRINTF(DEBUG_USER, "counter=%d", counter);
```

Notice: as a debug message in macro `CPPTESK_DEBUG_PRINT()` any "stream expression" (allowed for usage in standard C++ STL output streams expressions) can be used.

To add location information of debug macro to debug message (file name and string number) is possible by means of macros `CPPTESK_DEBUG_PRINT_FILE_LINE()` and `CPPTESK_DEBUG_PRINTF_FILE_LINE()`. Their parameters are the same as of macros mentioned above.

Process call stack printing

To print *process call stack* of reference model is possible by macro `CPPTESK_CALL_STACK()`, which can be used inside and instead of debug message of macro `CPPTESK_DEBUG_PRINT()`.

Example:

```
#include <hw/model.hpp>

CPPTESK_DEFINE_PROCESS(MyModel::some_process) {
    CPPTESK_START_PROCESS();

    CPPTESK_DEBUG_PRINT(DEBUG_USER, "Call stack is "
        << CPPTESK_CALL_STACK());
    ...
    CPPTESK_STOP_PROCESS();
```

```
}
```

Notice: macro `CPPTESK_CALL_STACK()` can be used only inside on reference model.

Colored debug print

To facilitate manual search of debug messages of a certain type among all debug print is possible by means of *colored debug print* macros `CPPTESK_COLORED_DEBUG_PRINT(level, color, background_color, message)`, `CPPTESK_COLORED_DEBUG_PRINTF(level, color, background_color, format, parameters)`, and also macros `CPPTESK_COLORED_DEBUG_PRINT_FILE_LINE()` and `CPPTESK_COLORED_DEBUG_PRINTF_FILE_LINE()`.

The following color constants are defined (namespace `cpptesk::hw`):

- `BLACK` — black;
- `RED` — red;
- `GREEN` — green;
- `YELLOW` — yellow;
- `BLUE` — blue;
- `MAGENTA` — purple;
- `CYAN` — cyan;
- `WHITE` — white.

Example:

```
#include <hw/debug.hpp>

using namespace cpptesk::hw;
...
CPPTESK_COLORED_DEBUG_PRINT_FILE_LINE(DEBUG_USER, RED, BLACK,
    "The input message is " << CPPTESK_GET_MESSAGE());
...
CPPTESK_COLORED_DEBUG_PRINTF(DEBUG_USER, WHITE, BLACK,
    "counter=%d", counter);
```

Controlling indents in debug print

To *control indents* in debug print is possible by `CPPTESK_SET_DEBUG_INDENT(indent)`, `CPPTESK_BEGIN_DEBUG_INDENT`, and `CPPTESK_END_DEBUG_INDENT` macros. Macro `CPPTESK_SET_DEBUG_INDENT` sets indent value (not negative integer) returning its old value. Macros `CPPTESK_BEGIN_DEBUG_INDENT` and `CPPTESK_END_DEBUG_INDENT` are used in complementary way: the first one increases indent, the second one decreases indent by one point.

Example:

```
#include <hw/debug.hpp>

using namespace cpptesk::hw;
...
unsigned old_indent = CPPTESK_SET_DEBUG_INDENT(2);
...
CPPTESK_BEGIN_DEBUG_INDENT
{
    CPPTESK_DEBUG_PRINT(DEBUG_USER, "Some message");
    ...
}
CPPTESK_END_DEBUG_INDENT
```

Debug print levels

There is a *level* of debug message parameter among other debug print parameters. Level characterizes importance of the message. Usually, debug messages of different levels are colored differently. The following debug levels are defined (namespace `cpptesk::hw`):

- `DEBUG_MORE` — detailed debug messages produced by toolkit itself;
- `DEBUG_INFO` — basic debug messages produced by toolkit itself;
- `DEBUG_USER` — user's debug messages;
- `DEBUG_WARN` — warnings (typically, produced by toolkit itself);
- `DEBUG_FAIL` — messages about failures (typically, produced by toolkit itself).

The most "*important*" level is `DEBUG_FAIL`, then `DEBUG_WARN`, etc. `DEBUG_USER` is the only one level for user's messages.

Debug print setting up

To set up the volume of debug print messages is possible by selection of *debug print level*, and only those messages will be printed, which has debug level being not less than selected one. It is done by macro `CPPTESK_SET_DEBUG_LEVEL(`*`debug_level, colored`*`)`. This macro has an additional Boolean parameter *`colored`*, turning on/off coloring. Debug level `DEBUG_INFO` is set by default. Special level `DEBUG_NONE` can be used to switch off debug print totally.

Each debug print level can be assigned with *colors* for messages of this level. It is done by macro `CPPTESK_SET_DEBUG_STYLE(`*`level, tag_color, tag_background_color, color, background_color`*`)`.

Example:

```
#include <hw/model.hpp>

using namespace cpptesk::hw;
...
// reference model constructor
MyModel::MyModel() {
    // print messages with failures only,
    // switch on message coloring
    CPPTESK_SET_DEBUG_LEVEL(DEBUG_FAIL, true);
    // [FAIL] Error message style.
    CPPTESK_SET_DEBUG_STYLE(DEBUG_FAIL, BLACK, RED, RED, BLACK);
}
```