# SystemC 2.1 Features

This document provides a list of the features and modifications in SystemC 2.1 over SystemC 2.0.1.

1. **Dynamic process creation**

   In addition to processes created in sc_module constructors via the SC_METHOD and SC_THREAD macros, processes can also be created after simulation starts with the sc_spawn() API. The implementation uses (and ships) a part of the publicly available boost library (www.boost.org). In particular the boost::bind templates are used. User code must define the macro SC_INCLUDE_DYNAMIC_PROCESSES before including "systemc.h" in order for the right header files to get included.

   Language Constructs
   ---
   The user-visible constructs for dynamic process creation and synchronization are:

   ```
   sc_spawn(...)
   sc_spawn_options(...)
   SC_FORK
   SC_JOIN
   sc_process_handle::wait()
   sc_bind(...)
   sc_ref(...)
   sc_cref(...)
   ```

   Basic Usage of sc_spawn
   ---

   Given the following function and method declarations:

   ```
   returnT my_function( ARGS );

   returnT my_class::my_method ( ARGS );
   ```

   To spawn these, use:

   ```
   returnT r;

   my_class* c;
   c = this; // or point to some other object...
   ```

```
  sc_process_handle h1 = sc_spawn( &r, sc_bind(&my_class::my_method, c,
ARGS ));
  sc_process_handle h2 = sc_spawn( &r, sc_bind(&my_function,          ARGS ));
```

Function Arguments
---

A spawned function can have up to 9 arguments, a spawned class method up to 8 arguments (this restriction comes with the usage of boost bind library).

Strict type checking of arguments is done. Arguments can by passed by value (default), per reference (use sc_ref) or per const reference (use sc_cref). Example:

```
    int my_function( double FA1, double &FA2, const double &FA2 );

    int r;
    double A;

    sc_spawn( &r, sc_bind(&my_function, A, sc_ref(A), sc_cref(A)) );
```

If the spawned function returns no value, or if you do not wish to use the returned value, the first argument (r above) may be omitted:

```
    sc_spawn( sc_bind(&my_function, A, sc_ref(A), sc_cref(A)) );
```

If the first argument is included, the pointed to space must be kept valid until spawned function completes, at which point the returned value will be stored in the space.

sc_spawn_options
---

After the sc_bind() argument to sc_spawn() is specified, two more optional arguments can be specified to sc_spawn(). The second of these two optional arguments is a pointer to sc_spawn_options. sc_spawn_options can be used to control the spawning of a thread process or a method process, and to specify static sensitivity information and dont_initialize information for dynamic spawned processes, similar to static processes. For spawned threads, the stack size information can also be specified through sc_spawn_options. The sc_spawn_options class supports the following API:

void set_stack_size(int stack_size);

// specify stack size for threads, ignored for methods

void spawn_method();
// spawn a method process, the default is a thread process

void dont_initialize();
// don't schedule the spawned process for an initial execution,
// by default it is scheduled for an initial execution

void set_sensitivity(sc_event* e);
// make spawned process statically sensitive to the event

void set_sensitivity(sc_port_base* p);
// make spawned process statically sensitive to the default event
// of the interface bound to the port

void set_sensitivity(sc_interface* i);
// make spawned process statically sensitive to the default event
// of the interface

void set_sensitivity(sc_event_finder* f);
// make spawned process statically sensitive to the event
// returned by the find_event() member of the event finder

Each of the set_sensitivity() methods can be called multiple times to indicate static sensitivity on multiple objects (e.g., specify sensitivity on events e1 and e2, and port p1).

Naming a spawned process:
---

The first of the 2 optional arguments that sc_spawn() accepts after the sc_bind() argument is – "const char* proc_name". The "proc_name" argument can be provided to name the spawned process. A spawned process gets a hierarchical name similar to other sc_objects. If the user explicitly provides a name "proc_name" to sc_spawn(), the full name of the spawned process is "parent_name.proc_name", where "parent_name" is the full name of the parent sc_object that spawned the process. Note that, since a sc_spawn_options* argument must be provided in order to spawn a method process, a spawned method process must also be explicitly named by the user, otherwise the design won't compile. If the user spawns a thread process, and does not specify an explicit name, then the tool generates a name of the form "thread_p_N" where "N" is a number that indicates this is the "Nth" child thread of the direct parent, where names are not reused when children of the direct parent die. Note that if a currently executing process spawns another process, then the currently executing process is the direct parent of the spawned process.

Synchronization
-------

A Fork/Join construct is provided:

```
SC_FORK
    sc_spawn(...) ,
    sc_spawn(...) ,
    ...
SC_JOIN
```

Please note that individual sc_spawn(...) sections are separated by commas and that there is no curly braces ("{","}") used, nor a semicolon at the end of SC_JOIN.

The code will only wait until all spawned processes have returned.

It is also possible to wait for an individual spawned process to finish with the sc_process_handle::wait()  function:

```
sc_process_handle h = sc_spawn(...);
...
h.wait();
```

Note that SC_FORK/SC_JOIN as well as sc_process_handle::wait() indirectly calls wait(some_event) and therefore can only be used within a thread context. If you call SC_FORK/SC_JOIN within a method context or outside any process, then SystemC will produce a runtime error.

It is also an error to call sc_process_handle::wait() on a handle associated with a spawned method process, because a method process never finishes. Similarly, SC_FORK-SC_JOIN cannot be used to spawn any method process.

sc_spawn() merely creates a process and schedules it for an initial execution (unless dont_initialize is specified through sc_spawn_options) – it does NOT execute the process. The spawned process executes when control goes back to the scheduler.

It is important to note that sc_spawn is a strict superset of the functionality available via the SC_THREAD and SC_METHOD macros. The SC_THREAD and SC_METHOD macros are retained for compatibility with earlier versions of SystemC. However in SystemC 2.1 and in future versions of SystemC, it is not possible to invoke the SC_THREAD and SC_METHOD macros after simulation starts. In addition, it IS possible to call sc_spawn both before, and after simulation starts.

Example
---

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include <systemc.h>

int function_method(double d)
{
  cout << endl << sc_time_stamp() << ", "
       << sc_get_curr_process_handle()->name()
       << ": function_method sees " << d << endl;
  return int(d);
}

class module1 : public sc_module
{
private:
  sc_event& ev;
public:

  SC_HAS_PROCESS(module1);

  module1(sc_module_name name, sc_event& event) : sc_module(name),
    ev(event)
  {
    int r;
    SC_THREAD(main);
    cout << endl << sc_time_stamp() << ": CTOR, Before spawning
function_method" << endl;
    sc_spawn_options o1;
    o1.spawn_method();
    o1.dont_initialize();
    o1.set_sensitivity(&ev);
    sc_process_handle h4 = sc_spawn(&r, sc_bind(&function_method, 1.2345),
"event_sensitive_method", &o1);

  }

  void main()
  {
    sc_event e1, e2, e3, e4;
```

```
    cout << endl << sc_time_stamp() << ", "
    << sc_get_curr_process_handle()->name()
    << ": main thread, Before spawning round robin threads."
    << endl << endl;

    e1.notify(100, SC_NS);

    // Spawn several threads that co-operatively execute in round robin order

    SC_FORK
      sc_spawn(
        sc_bind(&module1::round_robin, this, "1", sc_ref(e1), sc_ref(e2), 3), "1") ,
      sc_spawn(
        sc_bind(&module1::round_robin, this, "2", sc_ref(e2), sc_ref(e3), 3), "2") ,
      sc_spawn(
        sc_bind(&module1::round_robin, this, "3", sc_ref(e3), sc_ref(e4), 3), "3") ,
      sc_spawn(
        sc_bind(&module1::round_robin, this, "4", sc_ref(e4), sc_ref(e1), 3), "4") ,
    SC_JOIN

    cout << endl << sc_time_stamp() << ", "
         << sc_get_curr_process_handle()->name()
         << ": Done main thread." << endl;
  }

  void round_robin(const char *str, sc_event& receive, sc_event& send, int cnt)
  {
    while (--cnt >= 0)
    {
      wait(receive);
      cout << sc_time_stamp() << ": " << sc_get_curr_process_handle()->name()
           << ": Round robin thread " << str << endl;
      wait(10, SC_NS);
      send.notify();
    }
  }

};

int sc_main (int argc , char *argv[])
{
  sc_event event1;
  event1.notify(55, SC_NS);

  module1 mod1("mod1", event1);
  sc_start(500, SC_NS);
```

```
  return 0;
}
```

2. **sc_export**

exports are an addition to ports and allow to export an interface through the module hierarchy. The export makes an interface -that is bound to a channel located somewhere within that module- available to the outside of the module. If you see a module with an export then you can be sure that this module already has a channel bound to this export.

Binding
---

Exports are similar to ports with respect to binding. An export can be bound to either a channel or another export, given that this export itself is directly or indirectly bound to a channel. Types must match which is checked either during compilation or elaboration

Binding can be done by name, or by CTOR. Binding is generally done just like ports, except for the following:

If you bind an export to another export like port E.IFP2 to D.IFP in the example below, then you must bind the parent_exp to the child_exp, e.g. "parent_exp(child_exp)". For ports, this is generally done the opposite way like child_port(parent_port), however this is wrong for exports and leads to an error during elaboration.

As a rule of thumb, bind "further(closer)" with "further" the port/export that is further away from the channel. This further(closer) rule works for ports as well as exports. For hierarchical ports, the channel is connected to the port of the top-most module, so parent_port==closer, hence do a child_port(parent_port). For an export, the channel is embedded to the innermost instance, so child_exp==closer, hence do parent_exp(child_exp)

Names
---

An export can be given an explicit name through the CTOR. If not, then a default name like "export_0", "export_1", ... is given with an individual number set for each module.

Supported Functions and Restrictions
---

You can access the interface of an export with the get_interface() method as well as with operator ->.

It is not allowed to use an export as an argument in the sensitivity list of a process. Furthermore, exports are not allowed in lambda expressions. Processes can use ports but not exports in these contexts.

Example
---

In this example, module D contains a channel of type C which implements an interface C_if. D makes the interface C visible to the outside by an export named "IFP". Module E contains an instance of D and also contains another instance of C. E exports both interfaces as exports IFP1 and IFP2. Both IFP1 and IFP2 are bound to ports P1 and P2 of module X.

```
// Interface
class C_if : virtual public sc_interface
{
public:
   virtual void run() = 0;
};

// Channel
class C : public C_if, public sc_channel
{
public:
   SC_CTOR(C) { }
   virtual void run()
   {
      cout << sc_time_stamp() << " In Channel run() " << endl;
   }
};

// --- D: export channel C through IFP --------
SC_MODULE( D )
{
```

```
    sc_export<C_if> IFP;
    SC_CTOR( D )
        : IFP("IFP"),   // explicit name
          m_C("C")
    {
        IFP( m_C );     // bind sc_export->interface by name
    }
 private:
    C m_C;            // channel
};

// --- E: module with two interface-ports ---
SC_MODULE( E )
{
 private:
    C m_C;
    D m_D;
 public:
   sc_export<C_if> IFP1;
   sc_export<C_if> IFP2;

    SC_CTOR( E )
        : m_C("C"),
          m_D("D"),
          IFP1("IFP1", m_C)
    {
        IFP2( m_D.IFP );          // bind sc_export->sc_export by name
    }
};

// Module X connected to the channels through E
SC_MODULE( X )
{
    sc_port<C_if> P1;
    sc_port<C_if> P2;
    SC_CTOR(X) {
        SC_THREAD(run);
    }
    void run() {
        wait(10, SC_NS);
        P1->run();
        wait(10, SC_NS);
        P2->run();
    }
};
```

```
int sc_main(int argc, char** argv) {
  E the_E("E");
  X the_X("X");
  // port->IFP
  the_X.P1( the_E.IFP1 );
  the_X.P2( the_E.IFP2 );

  sc_start(17, SC_NS);
  return 0;
}
```

## 3. Exception reporting API

The exception reporting facility provides a common and configurable API to report an exceptional situation.

The facility is presented by two classes: sc_report_handler and sc_report.
The former provides configuration and report generation calls.
The latter just contains the report related information.

The application defines an exceptional situation by using one of SC_REPORT_ macros to generate a report. The report is identified by its severity (represented by sc_severity enum type) and the message type. The message type is a string of characters, uniquely identifying a specific type of the exception.

This sc_severity describes the severity of a report:

```
enum sc_severity { SC_INFO, SC_WARNING, SC_ERROR, SC_FATAL };
```

SC_INFO The report is informative only.

SC_WARNING The report indicates a potentially incorrect condition.

SC_ERROR The report indicates a definite problem during execution. The default configuration forces a throw of a C++ exception sc_exception with the corresponding report information attached.

SC_FATAL The report indicates a problem which cannot be recovered from. In default configuration, the simulation is terminated immediately using an abort() call after reporting a SC_FATAL report.

The application can define actions to be taken for a generated report.

Whereas a usual reaction on a exceptional situation includes just printing a message, more complex scenarios could involve a logging of the report into a file, throwing a C++ exception or drop in the debugger.
The enum type sc_actions describes such a set of operations.

There are several predefined values for this type:

```
enum {
   SC_UNSPECIFIED  = 0x00,
   SC_DO_NOTHING   = 0x01,
   SC_THROW        = 0x02,
   SC_LOG          = 0x04,
   SC_DISPLAY      = 0x08,
   SC_CACHE_REPORT = 0x10,
   SC_INTERRUPT    = 0x20,
   SC_STOP         = 0x40,
   SC_ABORT        = 0x80
};
```

SC_UNSPECIFIED Take the action specified by a configuration rule of a lower precedence.

SC_DO_NOTHING Don't take any actions for the report, the action will be ignored, if other actions are given.

SC_THROW Throw a C++ exception (sc_exception) that represents the report.

SC_LOG Print the report into the report log, typically a file on disk. The actual behavior is defined by the report handler function described below.

SC_DISPLAY Display the report to the screen, typically by writing it in to the standard output channel using std::cout.

SC_INTERRUPT Interrupt simulation if simulation is not being run in batch mode. Actual behavior is implementation defined, the default configuration calls sc_interrupt_here(...) debugging hook and has no further side effects.

SC_CACHE_REPORT Save a copy of the report. The report could be read later using sc_report_handler::get_cached_report(). The reports saved by different processes do not overwrite each other.

SC_STOP Call sc_stop(). See sc_stop() manual for further detail.

SC_ABORT        The action requests the report handler to call abort().

The report handler, a function known to the class sc_report_handler, takes the responsibility of execution of the requested actions. Application is able to redefine the report handler to take additional steps on execution of a specific action or extend the default set of possible actions.
As the report handler is responsible for all predefined actions it can also be used to redefine the behavior of predefined actions.

Each exception report can be configured to take one or more sc_actions. Multiple actions can be specified using bit-wise OR. When SC_DO_NOTHING is combined with any thing other than SC_UNSPECIFIED, the bit is ignored by the facility.
In addition to the actions specified within the sc_actions enum, via sc_actions, the exception API also can take two additional actions. The first action is always taken: the sc_stop_here() function is called for every report, thus providing users a convenient location to set breakpoints to detect error reports, warning reports, etc. The second action that can be taken is to force SC_STOP in the set of the actions to be executed. The action is configured via the stop_after() method described below, which allows users to set specific limits on the number of reports of various types that will usually cause simulation to call sc_stop().

The configuration and report generation API is contained within the sc_report_handler class.

The sc_report_handler class
----------------------------------

The class provides only static API. The user cannot construct an instance of the class.

```
  void report(
    sc_severity severity,
    const char* msg_type,
    const char* msg,
    const char* file,
    int         line
  );
```

Generate a report instance, which will cause the facility to take the appropriate actions based on the current configuration.
The call will configure a not known before exception of msg_type to take default set of actions for given severity.

The first occurrence of the particular msg_type starts its stop_after() counter.

```
sc_actions set_actions(
  sc_severity severity,
  sc_actions actions = SC_UNSPECIFIED
);
```

Configure the set of actions to take for reports of the given severity (lowest precedence match). The previous actions set for this severity is returned as the result. SC_UNSPECIFIED is returned if there was no previous actions set for this severity.

```
sc_actions set_actions(
  const char* msg_type,
  sc_actions  actions = SC_UNSPECIFIED
);
```

Configure the set of actions to take for reports of the given message type (middle precedence match). The previous actions set for this message type is returned as the result. SC_UNSPECIFIED is returned if there was no previous actions set for this message type.

```
sc_actions set_actions(
      const char* msg_type,
      sc_severity severity,
      sc_actions  actions = SC_UNSPECIFIED
);
```

Configure the set of actions to take for reports having both the given message type and severity (high precedence match). The previous actions set for this message type and severity is returned as the result. SC_UNSPECIFIED is returned if there was no previous actions set for this message type and severity.

The functions stop_after(...) modify only the limit, they do not affect the counter of the number of reports. Setting the limit below the number of already occurred reports will cause sc_stop() for the next matching report.

```
int stop_after(
  sc_severity severity,
```

```
    int limit = -1
  );
```

Call sc_stop() after encountering limit number of reports of the given severity (lowest precedence match). If limit is set to one, the first occurrence of a matching report will cause the abort. If limit is 0, abort will never be taken due to a matching report. If limit is negative, abort will never be taken for non-fatal error, and abort will be taken for the first occurrence of a fatal error. The previous limit for this severity is returned as the result. The stop_after() call will return UINT_MAX (int -1) in the case where no previous corresponding stop_after() call was made.

```
  int stop_after(
    const char* msg_type,
    int limit = -1
  );
```

Call sc_stop() after encountering limit number of reports of the given message type (middle precedence match). The previous limit for this message type is returned as the result. If limit is 0, abort will never be taken due to a matching report. If limit is negative, the limit specified by a lower precedence rule is used. The stop_after() call will return UINT_MAX in the case where no previous corresponding stop_after() call was made.

```
  int stop_after(
    sc_msg_type msg_type,
    sc_severity severity,
    int limit = -1
  );
```

Call sc_stop() after encountering limit number of reports having both the given message type and severity (highest precedence match.) If limit is 0, abort will never be taken due to a matching report. If limit is negative, the limit specified by a lower precedence rule is used. The previous limit for this message type and severity is returned as the result. The call will return UINT_MAX in the case where no previous corresponding stop_after() call was made.

```
  sc_actions suppress(
        sc_actions actions
  );
```

Suppress specified actions for subsequent reports regardless of configuration and clears previous calls to suppress(). The return value is the actions that were suppressed prior to this call. The suppressed actions are still active if they are mentioned by force(sc_actions) call.

```
sc_actions suppress();
```

Restore default behavior by clearing previous calls to suppress(). The return value is the actions that were suppressed prior to this call.
The default behavior does not suppress any actions.

```
sc_actions force(
        sc_actions actions
);
```

Force specified actions to be taken for subsequent reports in addition to the actions specified in the current configuration and clears previous calls to force(). The return value is the actions that were forced prior to this call.
The actions given by this call override similar setting in suppress().

```
sc_actions force();
```

Restore default behavior by clearing previous calls to force(). The return value is the actions that were forced prior to this call.
There is no forced actions in the default configurations.

```
sc_actions get_new_action_id();
```

Return an unused sc_actions value. Returns a different value each time it is called (returns SC_UNSPECIFIED if no more unique values are available).
Used when establishing user-defined actions, interpreted by a non-default report handler.
It is implementation defined whether the call could be used in the global constructors.

```
const char* get_log_file_name();
```

Return the log file name currently in effect. Return NULL if no logging is active at the moment.
It is implementation defined whether the returned string actually represents a file.

```
void set_log_file_name(
        const char* name
);
```

Set the log file name. The current handler implementation is responsible for interpretation of the given argument. The name may be unused until first SC_LOG action has occurred.
The default implementation provides a plain text file logging. The file will be opened as part of the first SC_LOG action. The report handler is responsible for proper terminating of the logging facility at the end.

```
const sc_report* get_cached_report();
```

Return pointer to the recent report for which an SC_CACHE_REPORT action was defined. In the default configuration, reports of severities SC_ERROR and SC_FATAL are cached.

```
void clear_cached_report();
```

Clear cached report for the current process (if any).

```
void initialize();
```

Initializes default configuration.
The call shall reset the limit counters.
The call may not remove or reconfigure messages.
The call may not affect logging.
The call does not affect cached reports.

```
void release();
```

Releases the resource possibly allocated by the exception reporting implementation. The facility may not be used after this call. Whether the facility could be used after subsequent initialize() call is defined by the implementation. The default implementation removes all user defined and/or configured messages and closes the log file. Configured predefined messages will be not reset.

```
void set_handler(
        sc_report_handler_proc handler
);
```

```
typedef void (*sc_report_handler_proc)(const sc_report&, const sc_actions&);
```

Specify the report handler function. The handler functions get an instance of the generated report and can use the methods of sc_report to access the needed information. The set of requested actions is passed through the second argument.

```
void default_handler(
        const sc_report& report,
        const sc_actions& actions
);
```

The function is the default handler of the facility provided by the given SystemC implementation.

The force() and suppress() methods provide a brute-force way to override the current configuration. For example, force(SC_LOG) could be called during debugging to cause all reports to be logged regardless of the current configuration. As another example, suppress(suppress() | SC_THROW); could be called by code that is not C++ throw-safe when it starts execution, and then suppress(prev) would be called when it completes execution.

The class sc_report - the report representation.

An instance of the class could be accessed through its cached copy.
Use sc_report_handler::get_cached_report() to access the cached copy of the report.
Instances of the sc_report can be copied by copy constructor and assignment operator means. It is not allowed to create an empty report.

The sc_report class
-----------------------

```
sc_severity get_severity() const;
```

Return the severity of a report object.

```
const char* get_msg_type() const;
```

Get message type of a report object.
The returned string is guaranteed to persist until
sc_report_handler::release() is called.

```
const char* get_msg() const;
```

Get message contents of a report object.
The lifetime of the returned pointer is that of the report instance.

```
const char* get_file_name() const;
```

Get file name that generated report object.
Please see the definition of the SC_REPORT_ macros for the exact contents of the returned value.

```
int get_line_number() const;
```

Get line number that generated report object. See also: get_file_name().

```
sc_time get_time() const;
```

Get the simulation time when then report object was generated.

```
const char* get_process_name() const;
```

Get the name of the process that generated the report object.

When a report is logged to a file, the current simulation time and current process name will automatically be included within the report.

The implementation defines following actions in the default configuration:

| Severity | Actions |
|---|---|
| INFO | SC_LOG \| SC_DISPLAY |
| WARNING | SC_LOG \| SC_DISPLAY |
| ERROR | SC_LOG \| SC_CACHE_REPORT \| SC_THROW |
| FATAL | SC_LOG \| SC_DISPLAY \| SC_CACHE_REPORT \| SC_ABORT |

The error level reports are displayed by the default handler of sc_exception type exceptions.

The following macros are globally visible as part of the standard and should be used to generate reports:

```
#define SC_REPORT_INFO(msg_type, msg) \
```

```
sc_report_handler::report( SC_INFO, msg_type, msg, __FILE__, __LINE__ )
#define SC_REPORT_WARNING(msg_type, msg) \
sc_report_handler::report( SC_WARNING, msg_type, msg, __FILE__,
__LINE__ )
#define SC_REPORT_ERROR(msg_type, msg) \
sc_report_handler::report( SC_ERROR, msg_type, msg, __FILE__, __LINE__ )
#define SC_REPORT_FATAL(msg_type, msg) \
sc_report_handler::report( SC_FATAL, msg_type, msg, __FILE__, __LINE__ )
```

The following examples illustrates how the exception API might be custom configured and how reports are generated. Note that message types are best captured within one or more header files, where they are declared using #define macros. This technique insures that strings representing message types are only declared once and that any typos that might occur when message types are specified in the SC_REPORT_* macros are caught by the compiler.

```
#define PCI_RPT_PROTOCOL_EXCEPTION    "PCI Protocol Exception"
const char PCI_RPT_PROTOCOL_READ_RETRY[] = "PCI Read Retry";

int sc_main(int, char**)
{
   // stop after having seen 10 error-level reports
   sc_report_handler::stop_after(SC_ERROR, 10);

   // make the PCI_RPT_PROTOCOL_EXCEPTION error non-critical
   // Note that 10 this errors will still cause a stop, as
   // configured by previous statement.
   sc_report_handler::set_actions(PCI_RPT_PROTOCOL_EXCEPTION,
                                  SC_ERROR,
                                  SC_DISPLAY);

   // disable the report PCI_RPT_PROTOCOL_READ_RETRY
   sc_report_handler::set_actions(PCI_RPT_PROTOCOL_READ_RETRY,
SC_DO_NOTHING);

   sc_start(1, SC_MS);

   // allow the report PCI_RPT_PROTOCOL_READ_RETRY to be displayed
   sc_report_handler::set_actions(PCI_RPT_PROTOCOL_READ_RETRY,
SC_DISPLAY);

   sc_start(1, SC_MS);

   // PCI_RPT_PROTOCOL_READ_RETRY reports will now be configured to
```

```
   // SC_UNSPECIFIED. Therefore, a lower precedence rule applies and the
   // actions in SC_DEFAULT_..._ACTIONS will take effect for the report.
   sc_report_handler::set_actions(PCI_RPT_PROTOCOL_READ_RETRY);

   sc_start(1, SC_MS);
}

void foo()
{
   sc_time max_time(500, SC_NS);

   if (...)
         SC_REPORT_ERROR(PCI_RPT_PROTOCOL_EXCEPTION,
                           "PCI burst read exceeded max time limit of " +
max_time.to_string());

   if (...)
         SC_REPORT_INFO(PCI_RPT_PROTOCOL_READ_RETRY,
                        "PCI read retry at time " + sc_time_stamp().to_string());
}
```

The following example illustrates how reports using SC_CACHE_REPORT actions
can be accessed:

```
   ...
   sc_report_handler::set_actions(PCI_RPT_PROTOCOL_READ_RETRY,
                                  SC_INFO,
                                  SC_CACHE_REPORT|SC_LOG);
   ...

   void module::do_something()
   {
         if (...)
            SC_REPORT_INFO(PCI_RPT_PROTOCOL_READ_RETRY, "...");
   }
   void module::foo()
   {
         sc_report_handler::clear_cached_report();
         do_something();

         sc_report* rp = sc_report_handler::get_cached_report();

         if ( rp ) {
            cout << rp->get_msg() << endl;
         }
```

```
}
```

The following example illustrates how reports using SC_THROW actions can be accessed:

```
...
sc_report_handler::set_actions(PCI_RPT_PROTOCOL_EXCEPTION,
                               SC_ERROR,
                               SC_THROW);
...

void module::do_something()
{
      if (...)
         SC_REPORT_ERROR(PCI_RPT_PROTOCOL_EXCEPTION, "...");
}
void module::bar()
{
      try
      {
         do_something();
      }
      catch (const sc_exception & e)
      {
         cerr << e.what() << endl;
      }
}
```

## 4. sc_event_queue

The queue has a similar interface like an sc_event but has different semantics: it can carry any number of pending notifications. The general rule is that _every_ call to notify() will cause a corresponding trigger at the specified wall-clock time that can be observed (the only exception is when notifications are explicitly cancelled).

If multiple notifications are pending at the same wall-clock time, then the event queue will trigger in different delta cycles in order to ensure that sensitive processes can notice each trigger. The first trigger happens in the earliest delta cycle possible which is the same behavior as a normal timed event.

Adding event notifications: add an event to the event-queue with the notify() function. For example

```
sc_event_queue E ("E");
E.notify( 10,SC_NS );
```

will add an event to E scheduled to occur 10 ns from now.

Waiting for events: use the event queue like any other event, for example

```
SC_METHOD( proc );
sensitive << E;
```

You can cancel all events from the queue with function cancel_all().

sc_event_queue is implemented as a channel that implements the sc_event_queue_if interface and sc_event_queue_port is conveniently declared as a sc_port using the sc_event_queue_if interface.

Example
---

```
SC_MODULE(Rec) {
  sc_event_queue_port E;

  SC_CTOR(Rec) {
    SC_METHOD(P);
    sensitive << E;
    dont_initialize();
  }
  void P() {
    cout << sc_time_stamp()
         << ": P awakes\n";
  }
};

SC_MODULE(Sender) {
  sc_in<bool> Clock;
  sc_event_queue_port E;

  SC_CTOR(Sender) {
      SC_METHOD(P);
      sensitive_pos << Clock;
      dont_initialize();
  }
  void P() {
      // trigger in now (2x), now+1ns (2x)
```

```
    E->notify( 0, SC_NS );
    E->notify( 0, SC_NS );
    E->notify( 1, SC_NS );
    E->notify( 1, SC_NS );
  }
};

SC_MODULE(xyz) {
  SC_CTOR(xyz) {
    SC_THREAD(P);
  }
  void P() {
    wait(15, SC_NS);
    cout << sc_time_stamp()
         << ": xyz awakes\n";
  }
};

int sc_main (int argc, char** argv)
{
  sc_event_queue E("E");

  Rec R("Rec");
  R.E(E);

  sc_clock C1 ("C1", 20);
  sc_clock C2 ("C2", 40);

  xyz xyz_obj("xyz");

  // Events at 0ns (2x), 1ns (2x),   20ns (2x), 21ns (2x),   40ns (2x), ...
  Sender S1("S1");
  S1.Clock(C1);
  S1.E(E);

  // Events at 0ns (2x), 1ns (2x),   40ns (2x), 41ns (2x),   80ns (2x), ...
  Sender S2("S2");
  S2.Clock(C2);
  S2.E(E);

  // Events at 3ns, 5ns (2x), 8ns
  sc_start(10);
  E.notify( 5,SC_NS );
  E.notify( 3,SC_NS );
  E.notify( 5,SC_NS );
  E.notify( 8,SC_NS) ;
```

```
  // Events would be at 40ns, 43ns (2x), 44ns but all are cancelled
  sc_start(40);
  E.notify( 3, SC_NS );
  E.notify( 3, SC_NS );
  E.notify( 4, SC_NS );
  E.notify( SC_ZERO_TIME );
  E.cancel_all();

  sc_start(40);
  return 0;
}
```

## 5. Notification callbacks for simulator phases

There are three new callbacks provided via virtual methods for classes derived from sc_module, sc_port, sc_export, and sc_prim_channel. These callbacks will be invoked by the SystemC simulation kernel when certain phases of the simulation process occur. The new methods are:

void before_end_of_elaboration();

This method is called just before the end of elaboration processing is to be done by the simulator.

void start_of_simulation();

This method is called just before the start of simulation. It is intended to allow users to set up variable traces and other verification functions that should be done at the start of simulation.

void end_of_simulation();

If a call to sc_stop() had been made this method will be called as part of the clean up process as the simulation ends. It is intended to allow users to perform final outputs, close files, storage, etc.

It is also possible to test whether the callbacks to the start_of_simulation methods or end_of_simulation methods have occurred. The boolean functions sc_start_of_simulation_invoked() and sc_end_of_simulation_invoked() will return true if their respective callbacks have occurred.

## 6. Support for programs with their own main() function

SystemC version 2.1 simplifies creation of simulations where there is a need of customized main function. To make possible to define the main function use code like in the example below:

```
#include <systemc.h>

int main(int argc, char** argv)
{
   ... do something ...
   // pass the control to SystemC
   int exit_code = sc_main_main(argc, argv);
   ... do something more ...
   return errors ? ... : exit_code;
}
```

The call sc_main_main will perform normal SystemC processing. At the moment it is not possible to call sc_main_main multiple times. The user still has to provide sc_main function.

## 7. sc_argc() and sc_argv()

SystemC version 2.1 allows access to the startup arguments of a simulation run via the functions sc_argc() and sc_argv():

```
int sc_argc();
const char * const * sc_argv();
```

## 8. Heterogeneous concatenation

The ability to concatenate the long datatypes sc_biguint<W>, and sc_bigint<W> is provided. You no longer have to copy to and from sc_bv<W> instances. In addition, you may now use any combination of the following data types, or bit and part selects of these data types, in a concatenation:

a) sc_int<W>
b) sc_uint<W>
c) sc_bigint<W>
d) sc_biguint<W>

sc_bv<W> and sc_lv<W> still form a separate group for concatenation purposes.

## 9. sc_stop() semantics change

The semantics of sc_stop() has been tightened in 2.1. When invoke from a process, control always returns to the invoking process, and after the invoking process returns/suspends, the current delta cycle is either completed, or not, depending on the specified stop mode. The stop mode can be specified with the new function sc_set_stop_mode:

void sc_set_stop_mode( sc_stop_mode mode );

mode may have one of the following values:

SC_STOP_IMMEDIATE - stop immediately
SC_STOP_FINISH_DELTA - finish the current delta cycle

If the stop mode is SC_STOP_IMMEDIATE, no more processes are executed, and the update phase is not executed. If the stop mode is SC_STOP_FINISH_DELTA, all processes that can be run in the current delta cycle are executed, and the update phase of the current delta cycle is also executed before simulation stops, and control returns to sc_main(). The default stop mode is SC_STOP_FINISH_DELTA. When sc_stop() is invoked from one of the phase callbacks (e.g., start_of_simulation), the current phase is completed before simulation stops.

If the start_of_simulation callbacks have happened before, then sc_stop() also triggers the end_of_simulation callbacks just before control returns to sc_main().

## 10. API changes for process information

Two new process related API entries have been added, and the behavior of an existing one has been modified.

In 2.0.1, sc_get_curr_process_handle() would return the currently executing process after simulation starts, and the last created process before simulation starts. In 2.1, sc_get_curr_process_handle() still returns the currently executing process after start of simulation, but returns NULL if invoked before start of simulation. A new API entry is available to access the last created process handle before start of simulation.

sc_process_b* sc_get_last_created_process_handle();

Another API entry has been added to obtain the kind information of the currently executing process - SC_METHOD_PROC_, SC_THREAD_PROC_, or SC_CTHREAD_PROC_. SC_NO_PROC_ is returned if no process is currently executing.

sc_curr_proc_kind sc_get_curr_process_kind();

## 11. sc_start(0)

This is no longer equivalent to sc_start() or sc_start(-1), which implies simulate forever. sc_start(0) now finishes all the delta cycles at the current time and returns. This function was formerly performed by sc_cycle(0) which has now been deprecated.

12. **New warning and error messages**

After sc_stop() has been called, a call to sc_start() produces an error message. After sc_stop() has been called, another call to sc_stop() issues a warning message. sc_cyle() is deprecated and produces a warning message. Applications using sc_cycle() still work, however a warning message is generated:

Info: (I540) sc_cycle is deprecated: use sc_start(...) instead

If you do not want to replace sc_cycle() calls and if you also must make sure that the output is identical to previous SystemC releases, then you can suppress the message using the following call:

```
sc_report_handler::set_actions(
  SC_ID_SC_CYCLE_DEPRECATED_, SC_DO_NOTHING );
```

13. **Link-time detection between incompatible implementations**

Object files compiled with different vendors of SystemC will now error out at link time. Source code compiled against the 2.1 headers will result in object code which reference a set of global symbols which encode the version and vendor tags of the library it was compiled against. This will result in link time errors for objects which are linked with a library other than the one they were compiled against. The vendors of customized versions of SystemC library have to provide own tags in addition to the 2.1 tags, depending on whether the binary interface was changed. The interface can be found in src/systemc/kernel/sc_ver.h and src/systemc/kernel/sc_ver.cpp.

**14. Version number in a standard format**

The version of the SystemC library being executed may now be acquired in a standard machine readable format. The sc_release() function will return a character string specifying the release using the following syntax:

<major_no>.<minor>.<patch>-<vendor>

where:

<major_no> is the major release number, e.g., 2
<minor_no> is the minor release number, e.g., 1
<patch>    is the patch designation, e.g., 0

<vendor>   is a string designating the vendor, e.g., OSCI

**15. Posix thread support**

SystemC 2.1 contains a version of thread support based on Posix threads. To create a version of SystemC which uses Posix threads in place of quick threads use the gmake commands

```
gmake pthreads
gmake install
```

when creating SystemC.

To build a debug library use the gmake commands

```
gmake pthreads_debug
gmake install
```

To test the examples use the gmake command

```
gmake pthreads_check
```

16. **The support for ISDB output of the tracing information is removed**.