SCP : Reporting library
Mark Burton
Keep It Generic

Goal of the SystemC Common Practices is that you should be able to pick up a ‘component’ on it’s own and use it – without needing to take the whole library.
SCP : reporting

- **What did we want:**
  - Generic and independent of other library components (of course)
    - Wanted this, but ... Our current implementation has some dependencies.
    - At least try to ensure that there is no dependencies on other SCP components.
  - Work with SystemC (using sc_report_ under the hood)
  - A convenient user API (FATAL, INFO....)
  - Nice syntax like FMT (“the answer is {}.”, 42)
  - VERY LOW simulation cost (a single ‘if’ to determin whether to report or not)
  - Driven by ‘CCI’ configuration so its easy to switch on/off

- **Existing implementation here:**
Some examples….

- SCP_TRACE() "My trace message"
- SCP_INFO("thing")("A is {} b is {} together they are {}", 1, 1, 42);
- SCP_DEBUG(())("Devils inside")
- SCP_WARN(())("George, don’t do that");
- SCP_FATAL(SCMOD) "No Going Back";

...
Architecture stack

Marco “fluff”
SCP_INFO.....

Sc_report ..... 

Spdlog ‘backend’
Existing features : Basic logging

- SCP_TRACE() << "My trace message"
  - Print a trace out.

- SCPTRACE("The answer is {}", 42);
  - Print using {FMT}

- SCP_TRACE("my feature") << "Another trace"
  - Print a trace on a feature. (The feature can then be switched off/on)

- SCP_TRACE(SCMOD) << "More trace"
  - Convenience to use the current module name as the feature name
Existing features : Init/Config

Uses spdlog ‘under the hood’ (an external dependencies ☹️)

MANY options to configure the output

(uses sc_report as a ‘backend’)

If this isn’t called, a ‘default’ setup is provided.

scp::init_logging(
  scp::LogConfig()
  .logLevel(scp::log::DEBUG)
  .msgTypeFieldWidth(20)
  .fileInfoFrom(5)
  .logAsync(false)
  .printSimTime(false)
  .logFileName(logfile));
CLArgumentParser : Parse command line for --gs_luafile option (11 arguments)

LuaFile_Tool Constructor

--images-dir is an internal option used for testing. Do not make any assumptions on its behavior as it may change or even disappear in the future.

Option --gs_luafile with value /Users/mburton/work/tmp/qqvp/configs/fw/8540/bsp/qnx/conf.lua

Lua file command line parser: parse option --gs_luafile /Users/mburton/work/tmp/qqvp/configs/fw/8540/bsp/qnx/conf.lua

Read lua file '/Users/mburton/work/tmp/qqvp/configs/fw/8540/bsp/qnx/conf.lua'

Setting param platform.with_gpu to value false

Setting param platform.timeprinter.log_level1 to value 4

Setting param platform.qemu_inst.sync_policy to value "multithread-quantum"

QOM Device creation l2vic

QOM Device creation qct-qtimer

Reset

QOM Device creation v67-hexagon-cpu
Existing features : Feature loggers

- Avoid “lookup hash”

- SCP_LOGGER((my_logger));
  - Define the variable my_logger as a logger that can be used in an SCP_TRACE.
  - The default logger is (), and it’s name will be the current module name (AND the module class name!)
  - Loggers can be named to other strings – e.g. SCP_LOGGER((),"my_feature")

- SCP_TRACE((my_logger)) << “More trace”;
  - my_logger is a variable in the current context (it is an integer which carries the level of logging above which the logger will output).

  - There is also an ‘array’ mechanism to build an array of loggers.

  - This whole mechanism is somewhat ‘awkward’ but we have it because we can’t associate loggers with modules within ‘standard’ SystemC.
Without the ‘logger’, the macro expands and calls a std::hash/map to find if we’re logging. (quite expensive)

The logger is just an int…

Initialized on first use (e.g. with CCI log level)

Once initialized, all SCP_ calls will use the int.

(“Macro magic all collapses to "if (this->logger) sc_report…”")
Existing features : CCI configuration

- Each feature (from a logger or not) can be enabled/disabled using CCI
  My.module.feature.log_level=5

- The value sets the level above which logging will be enabled.

- A bunch of ‘matching’ rules makes enabling/disabling easier:
  - E.g. top.log_level sets the log level for everything below top.
  - *.b.log_level sets the log level for anything with ‘b’ under it.
Problem: everything else wants to use it!

- Every single other components needs to use some sort of reporting
- It’s horrible to have to go back to sc_report...

- So – the reporting library needs to go upstream !!!!
So – what do we need (revisited)

- Generic and independent of other library components (of course)
- Work with SystemC (using `sc_report` under the hood)
  - Maybe better to build `sc_report` over the top of a better interface?
- A convenient user API (FATAL, INFO….)
- Nice ‘FMT’ syntax (“the answer is {}.”, 42)
- VERY LOW simulation cost (a single ‘if’ to determin whether to report or not)
- Driven by ‘CCI’ configuration so its easy to switch on/off
  - ‘CCI’ isn’t in the kernel or an IEEE standard (yet) – it can be one way to set what is enabled/disabled, but we could have a ‘clean’ interface.

- One (non) discussion, by the time this goes into the SystemC standard, we would probably be moving to C++20, which already has FMT ‘built in’ (so no issues about external libraries).
- Spdlog is currently a ‘back end’ implementation, which does not need to be part of the standard.
Architecture stack

This....

Marco “fluff”
SCP_INFO.....

Sc_report .....

Spdlog ‘backend’

Or maybe . . .

Sc_report .....

Marco “fluff”
SCP_INFO.....

Spdlog ‘backend’

To Be Descided . . .
- PLEASE JOIN IN!

- ‘loggers’ that send to multiple feature logs?
- Can we remove some complexity?
- Is spdlog the right ‘back end’ – should we build sc_report ontop, or should we use sc_report as a back end?

- “step one” – separate the CCI mechanism from the loggers, so the loggers can be added to e.g. ‘sc_module’ (anywhere else? sc_object?)