2. Exercise of the Laboratory
Entwicklung interaktiver eingebetteter Systeme

1 Introduction

In the following, you will start to work with SystemC. For this purpose, an eclipse workspace has been prepared for you.

1.1 Eclipse Workspace of the Lab

Please open the workspace by starting eclipse and choosing the directory depicted in Figure 1 as your workspace directory. In this workspace, the following eight projects are examples of various aspects of SystemC.

1. 02_sysc_ex01-startup – Examples of the various call backs called by SystemC while the simulation is set up.
2. 02_sysc_ex02-event – Event instantiation and notification.
3. 02_sysc_ex03-SC_METHOD – Registering a method process in a SystemC module.
4. 02_sysc_ex04-SC_THREAD – Registering a thread process in a SystemC module.
5. 02_sysc_ex05-SC_CTHREAD – Registering a clocked thread process in a SystemC module.
6. 02_sysc_ex06-HWSIGNAL – Example usage of SystemC signals.
7. 02_sysc_ex07-FIFO – Example usage of SystemC FIFOs.
8. 02_sysc_ex08-FIXPOINT – Examples for working with fix point data types.

![Eclipse Launcher](image.png)

Figure 1: Opening the workspace in directory ~/02_sysc/workspace-syasc
You might use the code in these projects for implementation hints for the tasks ahead of you in this lab.

Finally, the hwsim project should implement the SystemC simulation environment shown in Figure 2. This environment contains a VideoSource that captures images from a USB camera, which will be given to you during the lab, and a VideoSink that displays the video stream in a window, e.g., as shown in Figure 3.

```
void Top::driveResetLow() {
  // Lower reset after 11 clock cycles
}
```

Figure 2: Architecture realized by the SystemC simulation environment when using a FIFO to connect the modules

Every simulation you perform during this lab will be performed by starting the hwsim simulator. This simulator can be compiled (cf. Figures 11a, 12a, 15a, and 17a) with different build configurations, i.e., setting some defines differently, which you will use to either use SystemC signals or SystemC FI-FOs for communication between the VideoSource and VideoSink module. Moreover, the interface used by the VideoSource and VideoSink module might even differ and, hence, you have to develop an adapter module that translates between these different modes of communication.

To continue, open the hwsim project in the workspace as shown in Figure 4.

1.2 Handling Eclipse

You are using Eclipse CDT (C/C++ Development Tooling) for development. In CDT, there are two different kinds of errors that Eclipse can indicate for your source code. Examples for these two kinds are shown in Figure 6. First, there are errors indicated by the C++ compiler, e.g., as
indicated by the red disk containing a white x to the left of the line number 60 in Figure 6a. Eclipse parses the output of the C++ compiler – shown in the CDT Build Console – to find where the C++ compiler reported errors and marks all source code locations where errors where reported with the symbol of a red disk containing a white x. Hovering over this symbol will bring up the error text from the build console. However, sometimes it is easier to directly look at the build console to determine what the first error was. This is important to detect what is the cause and what might be subsequent errors stemming from the original error. In any case, these errors must be fixed by you for the program to compile. However, the symbols denoting these errors will only be updated when you compile the project, i.e., select one of the four build configurations (see Figure 5) and compile the project.
Figure 6: Two different kinds of errors indicated by Eclipse stemming from (a) the C++ compiler and (b) the Indexer.
Second, there is the Eclipse Indexer. The Indexer is used by Eclipse for code completion and navigation. Errors from the Indexer are indicated by a red bug symbol, e.g., as seen to the left of the line number 49 in Figure 6b. These errors might not be real and only shown due to (i) too complex C++ code the Indexer can’t parse but that is nonetheless valid C++ code, (ii) misconfiguration of the Indexer, (iii) missing auto-generated code, e.g., line 47 in Figure 6b missing the config.h header, that has not yet been generated, or (iv) differences between what files are present in the file system and what files Eclipse knows of. Errors from the Indexer can in principle be ignored but might negatively affect Eclipse code completion and navigation.

Indexer errors due to (i) should not happen during this lab. However, due to a bug in Eclipse, the Indexer is misconfigured to use a fixed build configuration instead of the active build configuration you are currently working on. To change this, follow the steps in Figure 7. Please note that due to the Eclipse bug, this change will revert if you close the Eclipse project and you have to reapply the fix.

If you opened the hwsim project for the first time, the config.h header will still be missing as it is auto-generated when you compile the project. In fact, the whole directory where this header will reside, i.e., obj-6435../implementation, is still missing. This fact can be observed by examining the include directories that are used by the Eclipse Indexer as shown in Figure 8. Grayed out include directories are currently missing or might be present in the file system but Eclipse is currently unaware of them.

To generate these include directories, compile the project with the VIDEOBARABSTRACTION_FIFO build configuration as shown in Figure 5 or Figure 11a. After you have started the compilation – even if it did not succeed – the config.h header should be present in the directory obj-6435../implementation. However, the directory might not immediately appear in the Eclipse project. If the directory is still not visible in Eclipse, then there is a differences between what files are present in the file system and what files Eclipse knows of. To notify Eclipse that new files have appeared in the project, you have to refresh the project by following the steps 1 and
Figure 8: Missing auto-generated include directories

② in Figure 9. After having refreshed the project (you could also have selected the project and pressed F5), the directory should be visible in Eclipse, e.g., as shown in Figure 9 with ③. Finally, rebuild the index as shown in Figure 10 to take into account the effect of the config.h header.
Figure 9: Refresh the hwsim project to pick up new files in the project directory

Figure 10: Rebuild the index for the hwsim project
Task 1 (Simple VideoSource to VideoSink SystemC Example)

Project: hwsim
Files: implementation/Top.hpp
implementation/Top.cpp

In the following, you will realize a simple VideoSource to VideoSink filter chain as depicted in Figure 2. For this purpose, you open the project hwsim as is depicted in Figure 4.

All filters can be compile in two different modes: (i) FIFO abstraction and (ii) RTL abstraction level. Here, #if VIDEOBARABSTRACTION_VIDEOSOURCE == VIDEOBARABSTRACTION_FIFO as well as #if VIDEOBARABSTRACTION_VIDEOSOURCE == VIDEOBARABSTRACTION_RTL are used to guard FIFO and RTL abstraction levels for the VideoSource module, respectively. Equivalently, the define VIDEOBARABSTRACTION_VIDEOSINK is used to guard the abstraction level used by the VideoSink module. In the following, both VideoSource and VideoSink will use SystemC FIFOs to communication with each other, i.e., the con1 channel in the Top module should be a SystemC FIFO. However, before you can start the example, you have to modify the Top module in the following ways:

a) Edit Top.hpp and Top.cpp to define a constructor for the SystemC module. Run the example by following the steps depicted in Figure 11. The example should compile and run but, of course, do nothing.

b) Overwrite the before_end_of_elaboration method of the module. Program code that should output stuff like the following:
   top: before_end_of_elaboration at @0 s called!

c) Overwrite the end_of_elaboration method of the module. Program code that should output stuff like the following:
   top: end_of_elaboration at @0 s called!

d) Overwrite the start_of_simulation method of the module. Program code that should output stuff like the following:
   top: start_of_simulation at @0 s called!

e) Overwrite the end_of_simulation method of the module. Program code that should output stuff like the following:
   top: end_of_simulation at @4081389 ms called!

f) Define the driveResetLow method and register it as a SystemC method process in the constructor of the module. Don’t forget to use the SystemC macro SC_HAS_PROCESS to enable SystemC process registration. Program code that should output stuff like the following:
   top: driveResetLow at @0 s called!

g) Define the event resetEndEvent and notify it after 11 cock cycles of the master clock with frequency MASTER_CLOCK_MHZ. This define is defined by the header hwsim_config.h. Modify the registration of the driveResetLow method to be sensitive to the resetEndEvent and no longer sensitive to the implicit startup event. This should output stuff like the following:
   top: driveResetLow at @50457 ps called!

h) Define the sReset signal initialize it to true in the constructor and reset it to false in the driveResetLow method. Take care to name the signal “sReset” in the constructor.

i) Define the con1 FIFO channel. Take care to name the channel ”con1” in the constructor.

j) Instantiate the submodules VideoSource and VideoSink as videoSource and videoSink. These classes are defined in the headers VideoSource.hpp and VideoSink.hpp, respectively.
(a) Compiling the filter at FIFO abstraction

(b) Starting the VideoSource $\rightarrow$ VideoSink example

Figure 11: Running and understanding the VideoSource $\rightarrow$ VideoSink example setup

k) Name the submodules in the constructor of the Top module accordingly, i.e., give their constructor the names "videoSource" and "videoSink", respectively.

l) Next, connect their Reset input ports to the $s_{Reset}$ signal of the Top module.

m) Then, use the $con1$ FIFO channel to connect the $Video\_out$ port of the VideoSource module to the $Video\_in$ port of the VideoSink module. Finally, run and test the VideoSource to VideoSink example by following the steps depicted in Figure 11. This should result in a display window similar to the one depicted in Figure 3.
Task 2 (Using an RTL Signal for the Video Signal)

Project: hwsim
Files: implementation/Top.hpp
      implementation/Top.cpp

In the following, you will switch the video signal from FIFO abstraction to RTL abstraction by
compiling as shown in Figure 12a. Thus, the defines VIDEOBARABSTRACTION_VIDEOSOURCE and

![Compiling the filter at FIFO abstraction](image1)

![Starting the VideoSource → VideoSink example](image2)

Figure 12: Running the VideoSource → VideoSink example at RTL

VIDEOBARABSTRACTION_VIDEOSINK will be set to VIDEOBARABSTRACTION_RTL. Moreover, it is also
required to connect a clock to the Clock ports of the VideoSource and VideoSink module. Thus,
you will need to instantiate a master clock with frequency MASTERCLOCK_MHZ and connect it to
both ports. Finally, the con1 signal must be switched from a FIFO to a SystemC signal to result
in the architecture depicted in Figure 13.

![Architecture realized by the SystemC simulation environment when using an RTL signal to connect the modules](image3)

Figure 13: Architecture realized by the SystemC simulation environment when using an RTL
signal to connect the modules

However, please guarantee that the FIFO abstraction level still remains functional. Therefore,
guard your changes with #if VIDEOBARABSTRACTION_VIDEOSOURCE == VIDEOBARABSTRACTION_RTL,
respectively, \#if VIDEOBARABSTRACTION_VIDEOSINK == VIDEOBARABSTRACTION_RTL. In detail, you have to modify the Top module in the following ways:

a) Switch the con1 signal from a FIFO to a SystemC signal carrying a \texttt{sc\_dt::sc\_uint<32>} data type. This change should enable a compilation at RTL. However, the resulting simulator will not run as the Clock ports of the VideoSource and VideoSink modules are not bound, i.e., are dangling and not connected to a clock.

b) Instantiate a clock – use an \texttt{sc\_core::sc\_clock} instance for this – as the member variable \texttt{clk}. The frequency of the clock should be \texttt{MASTERCLOCK\_MHZ}. Moreover, take care to name the clock “\texttt{clk}” in the constructor.

c) Connect the clock to the Clock port of the VideoSource module. Don’t forget to guard your changes with \#if VIDEOBARABSTRACTION_VIDEOSOURCE == VIDEOBARABSTRACTION_RTL.

d) Connect the clock to the Clock port of the VideoSink module. Don’t forget to guard your changes with \#if VIDEOBARABSTRACTION_VIDEOSINK == VIDEOBARABSTRACTION_RTL. Finally, compile the simulator at RTL abstraction level and start the simulation (see Figure 12).
Task 3 (Adapter from RTL to FIFO Video Signals)

Project: hwsim
Files: implementation/Top.hpp
implementation/Top.cpp
implementation/VideoAdapterRTLToFIFO.hpp
implementation/VideoAdapterRTLToFIFO.cpp

For large designs, it is not unusual to have different parts of the design at different levels of abstraction. In our lab, parts of the design might be at behavioral abstraction level (FIFO) and other parts at RTL abstraction level. In order to enable a simulation containing parts of the design at different abstraction levels, you will realize adapters to connect the communication protocols that are used at these different levels. For example, an adapter for the video signal communication from the RTL abstraction level to the FIFO abstraction level is required for the architecture depicted in Figure 14.

![Diagram of video signal communication from RTL to FIFO]

**Figure 14**: Architecture if an adapter from an RTL signal to a FIFO channel is required

a) Create the class `VideoAdapterRTLToFIFO` realizing a SystemC module in the files `VideoAdapterRTLToFIFO.hpp` and `VideoAdapterRTLToFIFO.cpp`.

b) Create a constructor for the SystemC module `VideoAdapterRTLToFIFO`. The first parameter of the constructor should be the name of the SystemC module.

c) Create the ports `Clock`, `Reset`, `Video_in`, and `Video_out` for the module. Take care to name the ports "Clock", "Reset", "Video_in", and "Video_out", respectively, in the constructor.

The above steps will create a dummy adapter module that can be used in the `Top` module. Of course, it still has no implementation and, thus, usage of this module will result in no `VideoSink` output. However, this dummy adapter module can be used to test the changes you will make in the `Top` module to accommodate the adapter. In detail, you have to realize following changes in the `Top` module and guard them accordingly for the case that an adapter from RTL to FIFO is required:
d) Create a con2 SystemC FIFO channel carrying a VideoSignal. Take care to name the signal accordingly in the constructor.

e) Instantiate the member variable adapter for the submodule VideoAdapterRTLToFIFO defined in the header VideoAdapterRTLToFIFO.hpp. Take care to name the submodule accordingly in the constructor.

f) Connect its Clock input port to the clk master clock of the Top module.

g) Connect its Reset input port to the sReset signal of the Top module.

h) Connect its Video_in input port to the con1 signal of the Top module.

i) Connect its Video_out output port to the con2 signal of the Top module.

j) Adapt the connection of the Video_in input port of the VideoSink module to use the con2 channel instead of con1.

After the above given steps, the VideoSource to VideoSink example should compile and run (see Figure 15), but produce no VideoSink output due to the missing adapter implementation. Thus, you will realize this implementation next. For this purpose, consider again Figure 14.

(a) Compiling the filter at FIFO abstraction

(b) Starting the VideoSource —> VideoSink example

Figure 15: Running the RTL to FIFO adapter between VideoSource and VideoSink

There, the Video_in port of the adapter carries sc_uint<32> values. However, not on every positive clock edge of the master clock, i.e., low to high transition of the clock on the Clock port, a new pixel will appear on the Video_in port. This only happens if the pixel clock – carried in bit 17 of the sc_uint<32> value – has a low to high transition. For these cases only, you have to write a VideoSignal value to the Video_out port of the adapter. In detail, you have to realize following changes in the VideoAdapterRTLToFIFO module:

k) Use the SC_HAS_PROCESS macro to specify that the module VideoAdapterRTLToFIFO can have processes.

l) Declare the method convertRTLToFIFO and register it as a SystemC process in the constructor of the module. Decide for yourself which process type to use and what sensitivity is appropriate.

m) Implement the method convertRTLToFIFO. Check the functionality of the adapter by compiling the hwsim project with VIDEOBARABSTRACTION_RTLTOFIFO and starting the Video source to sink Example filter chain by following the steps depicted in Figure 15.
Task 4 (Adapter from FIFO to RTL Video Signals)

Project: hwsim
Files:  
implementation/Top.hpp
implementation/Top.cpp
implementation/VideoAdapterFIFOToRTL.hpp
implementation/VideoAdapterFIFOToRTL.cpp

Here, you will develop an adapter for the video signal communication from the FIFO abstraction level to the RTL abstraction level as shown in Figure 16. As previously, you will first create a dummy implementation of the VideoAdapterFIFOToRTL adapter by following the following steps:

a) Create the class VideoAdapterFIFOToRTL realizing a SystemC module in the files VideoAdapterFIFOToRTL.hpp and VideoAdapterFIFOToRTL.cpp.

b) Create a constructor for the SystemC module VideoAdapterFIFOToRTL. The first parameter of the constructor should be the name of the SystemC module.

c) Create the ports Clock, Reset, Video_in, and Video_out for the module. Take care to name the ports "Clock", "Reset", "Video_in", and "Video_out", respectively, in the constructor.

The above steps will create a dummy adapter module that can be used in the Top module. Of course, it still has no implementation and, thus, usage of this module will result in no VideoSink output. However, this dummy adapter module can be used to test the changes you will make in the Top module to accommodate the adapter. In detail, you have to realize following changes in the Top module and guard them accordingly for the case that an adapter from FIFO to RTL is required:

d) Switch the con2 signal from a FIFO to a SystemC signal carrying a sc_dt::sc_uint<32> data type. Moreover, the con1 channel should already be a SystemC FIFO.

e) Switch the adapter submodule instantiation to the VideoAdapterFIFOToRTL class defined in the header VideoAdapterFIFOToRTL.hpp.

f) Ensure that the Video_in input port of the VideoSink module uses the con2 channel instead of con1.

After the above given steps, the VideoSource to VideoSink example should compile and run (see Figure 17), but produce no VideoSink output due to the missing adapter implementation.

g) Use the SC_HAS_PROCESS macro to specify that the module VideoAdapterRTLToFIFO can have processes.
h) Declare the method `convertFIFOToRTL` and register it as a SystemC process in the constructor of the module. Decide for yourself which process type to use and what sensitivity is appropriate.

i) Implement the method `convertFIFOToRTL`. Check the functionality of the adapter by compiling the `hwsim` project with `VIDEOBARABSTRACTION_FIFOTORTL` and starting the `Video source to sink Example` filter chain by following the steps depicted in Figure 17.

Figure 17: Running the FIFO to RTL adapter between `VideoSource` and `VideoSink`