# Loopy. Tutorial 1

Vladimir Rybalkin - rybalkin [at] rhrk.uni-kl.de

The purpose of this tutorial is to explain how to properly build and run the driver generator; compile client application. This includes generation of board-side and host-sided drivers; compiling client application running on the host. For profound understanding of the material of the current tutorial we highly encourage you to use "HOPP Driver Generator Documentation" [1], which was used as a base for the current tutorial.

1) **Short introduction.** This part explains main terms used in the tutorial.

a) **Driver.** The complete software product is called the driver. It enables communication between software running on the computer and the hardware platform on the board.

b) **Board / board-side.** The driver is split in two parts, one of which has to be uploaded and executed on the hardware platform itself. This part of the driver is referred to as board-side driver. Sometimes, the terms *server* and *server-side* might be used instead, since this driver acts similar to a server.

c) **Host / host-side.** In contrast to the board-side driver, the host-side driver is located on the communicating computer. This part contains the actual API, embedded developers will work with. Sometimes, the terms *client* and *client-side* might be used instead, since this driver part acts more like a client.

The overall architecture of the driver is depicted on Figure 1. Data is sent from an embedding host application to the host-side driver. This driver communicates over some transport medium with the board-side driver, which in turn distributes the received data to corresponding hardware components on the FPGA. Results are sent back through the same chain.

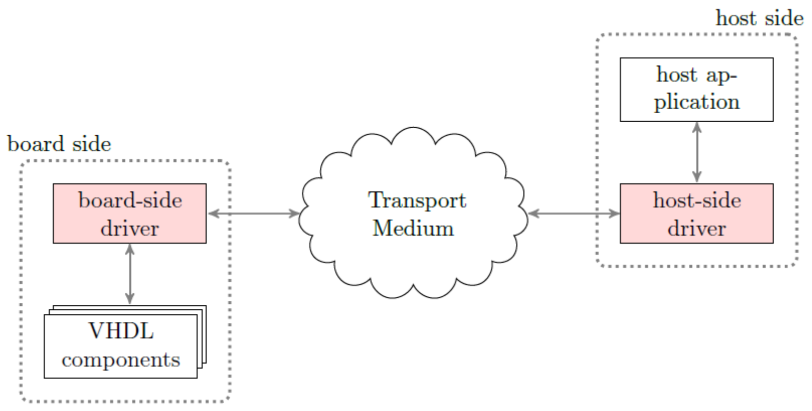


Figure 1: A high-level view of the data flow from a host application to hardware components on the board through the generated driver

2) **Setup.** In the following, the tools required to build and execute the driver generator are introduced. Please note that all tools have to be executable from the command line.

a) **Java.** The generator is implemented in Java. Consequently, JDK version 6 or higher is required. Run *"javac –version"* command in order to check the JDK version. The sample output *"java version 1.6.0_14"* signifies that JDK version 6 is installed.

b) **Gradle.** Gradle is a build tool required for building the driver generator. In order to be able to run gradle from any place, add the directory of the tool to *PATH* environment variable. Use *"setenv PATH ${PATH}:/software/gradle-1.5/bin".* In the case of *finance.eit.uni-kl.de* server use *"export PATH=$PATH:/opt/gradle-1.6/bin".*

c) **Xilinx Toolsuit.** In order to generate a *.bit* and an *.elf* files that are used to program an FPGA, the Xilinx toolsuite is required. Both XPS and the Xilinx SDK have to be accessible from console using the commands *"xps"* and *"xsdk"* respectively. In order to be able to invoke the tools from command line, use *"source /software/sources/XACT.rc/ise14.4".* You can skip this step, if you use *finance.eit.uni-kl.de* server.

d) **Mercurial.** Mercurial is a distributed versioning tool, comparable with GIT, Bazar, or (to some degree) Subversion. The sources of the driver generator are located in a mercurial repository. Check if the mercurial is installed by running *"hg"* command in the console.

e) **GCC.** GCC version 4.7.2 or higher is required. Run *"gcc --version"* command to find out the current version of the tool.

3) **Checkout.** The project has to be checked out. As of now, the project is only available at the mercurial repository of the Softech Group of the University of Kaiserslautern. The complete command for the initial checkout is *"hg clone https://softech.informatik.uni-kl.de/hg/public/loopy/ <target dir>"* Later the target directory is referred as a root directory.

4) **Building the driver generator.** After checking out the project, all sources have to be compiled and packaged. This can be done using the gradle build tool. The command to build the jar package is *"gradle jar".* Simply type it in a shell in the project root directory, where the

file *build.gradle* is located. Running the build file will generate an executable jar package under the path *<root dir>/build/libs.*

5) **Running the generator.** The behavior of the driver itself is only influenced by the provided *.bdf* file. Before running the generator, the board description file has to be modified. Navigate to *<root dir>/examples/AxiTwoRNG* directory and modify *system.bdf* file. The entries in the *medium ethernet* block have to be changed with respect to the FPGA board and network settings. The sample configuration:

```
medium ethernet {
mac “00:0a:35:02:31:95”
ip “172.16.13.146”
mask “255.255.0.0”
gate “172.16.254.254”
port 8844}
```

*a)*

```
medium ethernet {
mac “00:0a:35:02:31:95”
dhcp
port 8844}
```

*b)*

*a) the sample configuration in the case of the local network b) the sample configuration when DHCP settings are used (in the case of finance.eit.uni-kl.de* server*)*

In the current directory you have to find *build.sh* script that is used to run the driver generator. In order to make the script executable run command *“chmod +x build.sh”.* Then run *“sh ./buil.sh”* command which executes the script. In order to see the progress of the running driver generator, add verbose flag *“-v”* to *build.sh* script after a name of jar file. After successful completion of the process, you have to be able to find *temp*, *host* and *board* folders in the current directory. The *board* folder has to contain *system.bit* and *app.elf* files representing the board-side driver, which will be used later for an FPGA programming. The *host* directory has to contain */src* folder that includes all the files required for development of the client application running on the host.

6) **Building the client application.** The next step would be writing a host application that uses the generated host-side driver and API.

a) In the current project, we recommend to use Eclipse IDE for host side application development. Run *“eclipse”* command. Choose a path for workspace directory.

b) Create new project. *File –> New –> C++Project*. Specify the name for the project, for example, *“client_host_app”.* Choose the type of the project, *“Hellow World C++ Project”* with *“Linux GCC”* compiler as a toolchain.

c) Navigate to *<root dir>/examples/AxiTwoRNG/host/src* and copy all content to *<workspace dir>/client_host_app/src* directory of the eclipse project. Next, replace the content of the *“client_host_app.cpp”* with *<root dir>/examples/AxiTwoRNG /main.cpp.*

d) If DHCP settings are used, provide *startup()* function with IP address. Like this, *startup(“131.246.74.3”).*

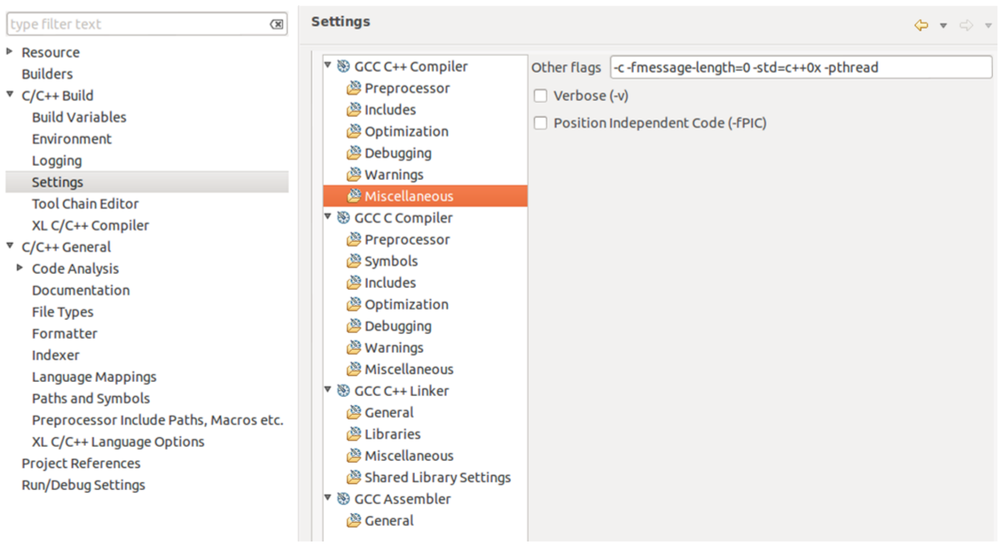
e) Apply the following settings for the project:

Do not forget to click apply after every step.

- Choose: *Project –> Properties.*
- Navigate to *C/C++ Build –> Settings –> GCC C++ Compiler –> Miscellaneous* and add *"-std=c++0x -pthread"* to *Other flags* textfield (see Figure 2). Add the same line to *C/C++ Build –> Settings –> GCC C++ Linker –> Miscellaneous –> Other flags.*
- Now, navigate to *C/C++ General –> Preprocessor Include Paths –> Providers*
- disable everything but *"CDT GCC Build-in Compiler Settings"*
- choose it and disable *"Use global provider shared between projects"* checkbox
- add *"-std=c++0x"* to *"Command to get compiler specs"* textfield that is now active (see Figure 3).

f) Left click on the project name in *"Project Explorer"* then *Project –> Build Project*. If you see errors, do *Project –> Clean...* then build project again.

g) Now, the project is built and can be run.

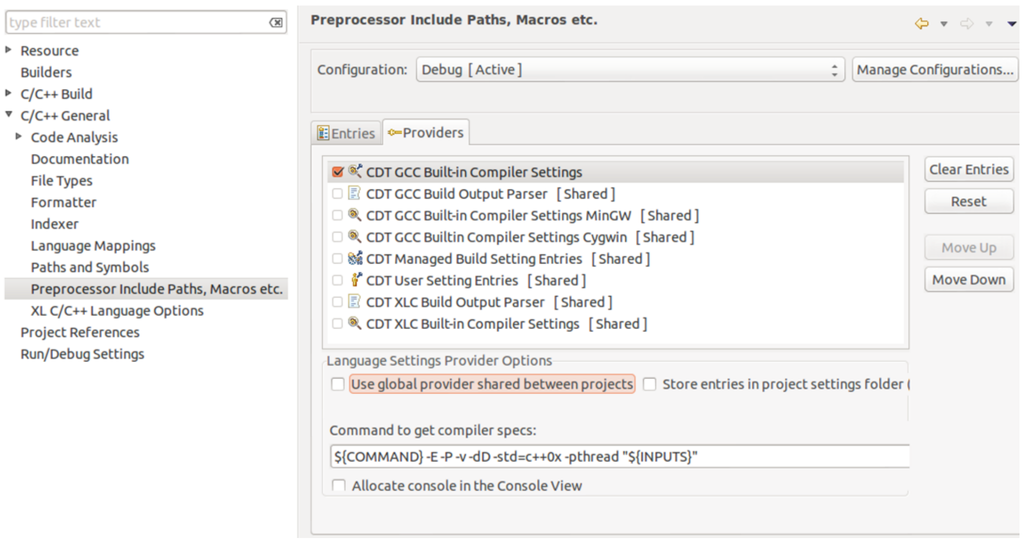


Other flags: -c -fmessage-length=0 -std=c++0x -pthread

☐ Verbose (-v)

☐ Position Independent Code (-fPIC)

Figure 2: Project settings



Configuration: Debug [ Active ]

- ✓ CDT GCC Built-in Compiler Settings
- ☐ CDT GCC Build Output Parser [ Shared ]
- ☐ CDT GCC Built-in Compiler Settings MinGW [ Shared ]
- ☐ CDT GCC Builtin Compiler Settings Cygwin [ Shared ]
- ☐ CDT Managed Build Setting Entries [ Shared ]
- ☐ CDT User Setting Entries [ Shared ]
- ☐ CDT XLC Build Output Parser [ Shared ]
- ☐ CDT XLC Built-in Compiler Settings [ Shared ]

Language Settings Provider Options

☐ Use global provider shared between projects ☐ Store entries in project settings folder (

Command to get compiler specs:

${COMMAND} -E -P -v -dD -std=c++0x -pthread "${INPUTS}"

☐ Allocate console in the Console View

Figure 3: Project settings

7) **Download HW/SW to the FPGA.**

a) Download the bitstream file *"system.bit"* and executable *"app.elf"* from the server to your host computer. Remember that these files are in *the <root dir>/examples/ AxiTwoRNG /board* folder.

b) Move the files to a handy location in your host computer, for example, your home folder.

c) Open a terminal there and type *"XMD"* and press enter. You should now be in the MicroBlaze Debug Module.

d) Type *"fpga -f system.bit"* and press enter, this will download the bitstream file to the FPGA.

e) Type *"connect mb mdm"* and press enter, this will connect the MicroBlaze to the Debug Module. In order to proceed, type *"rst"* and press enter.

f) Type *"dow app.elf"* and press enter; this will download the executable to our system.

g) Type *"rst"* and press enter, in order to reset the MicroBlaze.

h) Type *"con"* and press enter; this will start the software program.

i) Run client application by navigating to *<workspace dir>/client_host_app/Debug* and typing *"./client_host_app"* in the terminal.

j) Then, you should receive the first ten random values.

k) You could stop the MicroBlaze by typing *"stop"* in the *XMD* environment. Also, you can find more useful commands in the MicroBlaze documentation.

l) The following information is to speed-up the programming procedure. You could copy all these commands into a file and after initializing XMD, you can *perform "source myfile.txt".*

References:

[1] Thomas Fischer, "HOPP Driver Generator Documentation", Software Technology Group of the University of Kaiserslautern in cooperation with the Microelectronic Design Research Group of the University of Kaiserslautern, July 4, 2013