HOPP Driver Generator Documentation

Thomas Fischer

January 8, 2013

Contents

Chapter 1

Introduction

This document is intended to give an overview over the HOPP Driver Generator for both, users and developers. Users can find an easy to understand explanation of the generator as well as usage examples in Chapter 2. Developers and other interested readers can find details about the architecture and implementation of the generator in Chapter 3.

The HOPP Driver Generator is intended to provide embedded system developers with a simple, convenient interface to communicate with their designed hardware components.

This sentence is just some test for bibliography management... [1].

Chapter 2

User-Documentation

The purpose of this section is to explain new users how to use the generator as well as interact with the generated drivers.

2.1 Getting started

This part should contain all steps required in order to run the generator and acquire drivers for the specified board.

2.1.1 Building the project

First of all, 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.[1]

After checking out the project, all sources have to be compiled and packaged. This can be done using the gradle build tool.[2] Running the build file will generate an executable jar package under the path <project>/build/lib. Navigate there and try running it using the command java -jar <name of the jar package>.[3]

2.1.2 Command Line Interface

The HOPP Driver Generator can be called using the command line interface (CLI). While the tool generates C/C++ code, it is written in Java and therefore requires an installation of Java 6 or above. The CLI offers several parameters to futher configure the run of the generator, listed in Table 2.1.

[1] The repository is located at https://softech.informatik.uni-kl.de/hg/ag/hopp/. Please note, that authorization is required.

[2] Gradle is available at http://www.gradle.org/. The command to build a jar package is gradle jar. Simply type it in a terminal in the project root repository, where the file build.gradle is located. Also, JDK 6 or above is required to build the project.

[3] This should result in an error, since no .mhs file has been specified.

`-d`	`--dest`	Specifies the destination directory. All files will be generated into the specified directory. If none is specified, the current working directory will be used instead.
	`--debug`	The driver will be generated with additional debug console output. This makes it easier to track errors in larger test cases.
	`--mac`	used to set the mac address of a possible Ethernet interface of the board. Notation: XX:XX:XX:XX:XX:XX, where each X marks a hexadecimal number (allowing lower as well as upper cases).
	`--ip`	used to set the ip address of a possible Ethernet interface of the board. Notation: X.X.X.X, where each X marks a decimal number ranging from 0 to 255. The same notation is required for the following two parameters.
	`--mask`	used to set the network mask of a possible Ethernet interface of the board.
	`--gw`	used to set the standard gateway of a possible Ethernet interface of the board.
	`--port`	used to set the communication port of a possible Ethernet interface of the board. Note that these five parameters are only contemporary and will be replaced by the new board description language (I hope).
`-h`	`--help`	Lists all CLI parameters and a short explanation. The generator will abort after parsing this parameter and not generate anything.

Table 2.1: Summary of currently possible CLI parameters

Additionally, the `.mhs` file of the system is required by the generator. So a complete call looks like `java -jar driverGenerator.jar [OPTIONS] <.mhs file>`.

2.1.3 Resulting Artifacts

The output of the generator consists of two groups of files. The first group contains all files that make up the board side of the driver, which have to be compiled to an `.elf` file by the Xilinx SDK. The second group are files for the client side, that can be used to wrap communication with the board and its components.

Board Part

For the board, only two relevant files are generated, a header and a plain c source file. Both files have to be imported in a new Xilinx SDK project started from an existing XPS project. The `.elf` file generated from this project then has to be uploaded to the target board.

Client Part

The client side consists of several cpp sources and headers files. These files can be imported in any client side project and be used to wrap communication with components on the board.

2.2 API

This section should provide the user with a conceptual overview of the client part of the driver (which should be the only piece of software, you get in contact with). For a more detailed documentation of the code and provided methods, Javadoc style comments are provided, which can be transformed into an html or tex representation similar to the Java API using *doxygen*[4].

Generating the API Maybe, generate a script to do this (bashscript should suffice... it's a single command) or (even better) include this generation step in a more high level script calling the generator

After downloading and setting up doxygen, you can generate the API for your specific driver using the command `doxygen config.cfg` in either the client or server folder (which will generate the client or server documentation respectively). These config files are generated together with the c code. As a result, a `doc` folder containing both html and tex descriptions will be generated.

[4] `www.doxygen.org/`

2.2.1 Component

For each port on the component, a method returning the corresponding port object will be generated.

2.2.2 Port

A port object contains methods for receiving or sending data, depending on the type of the port. Naturally, receiving ports only contain methods for receiving data and sending ports only for sending. Dual ports contain the methods of both. Ports are restricted to a specific data type, i.e. bitwidth.

Chapter 3

Developer-Documentation

This section is addressed to future developers of the Driver Generator and describes the architecture and implementation of the generator itself.

3.1 Overview

3.1.1 JFlex & Cup

This can be kept short with a reference to the JFlex/Cup documentation

3.1.2 Katja

The driver generator uses the Katja tool, developed by the Software Technology Group of the University of Kaiserslautern. This tool generates several data types[1]. To be more specific, it provides the AST build up by the CUP parser as well as a model of the C/C++ language described in detail in Section 3.3.1.

3.2 Architecture

The overall idea of the driver generator is described by Figure 3.1. The source `.mhs` file describing the system board design is first translated into an internal representation of the board, i.e. an abstract syntax tree. This AST is used as input for the generator, which in turn outputs all required header- and source files.

The (intended) architecture of the host software is depicted in Figure 3.2. The core concept of the host software is the *component*. Such a component is a designed hardware unit, which can have several *ports*, over which data can be

[1]Since these data types will be described using their Katja specifications, it is strongly recommended to read through the Katja specification provided in form of three technical reports at `https://softech.informatik.uni-kl.de/Homepage/Katja`

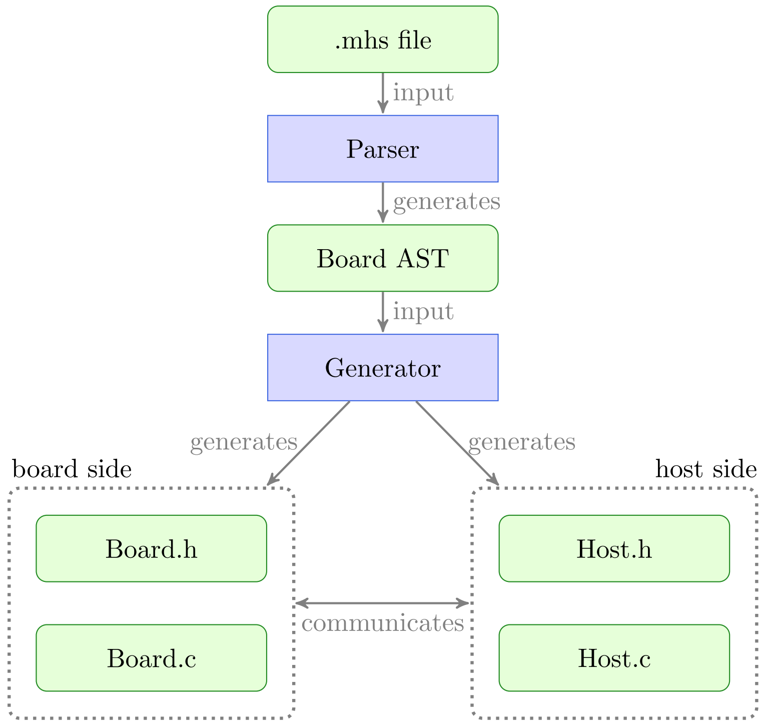

Figure 3.1: A rough sketch of the translation process so far

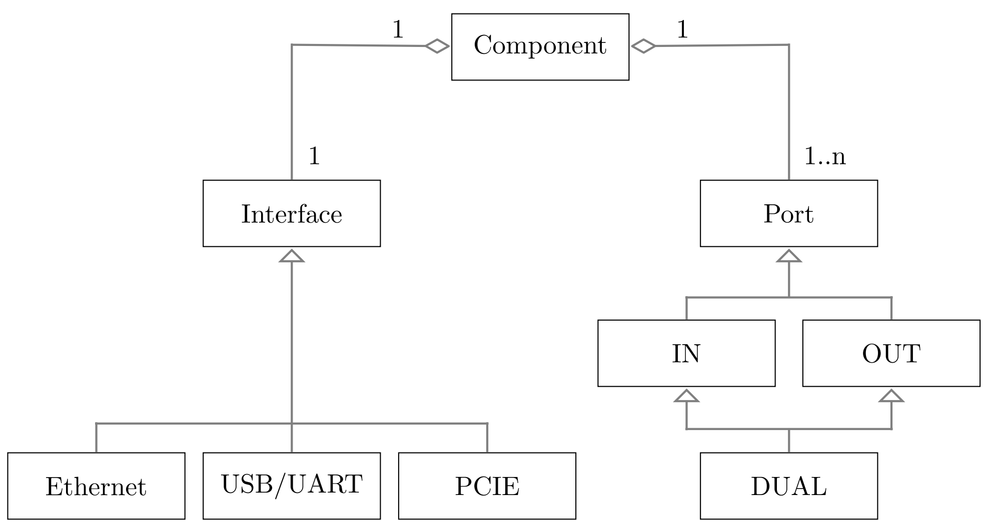

Figure 3.2: Architecture of the host part (to come)

sent to or received from the component. Hidden from the user, a component also has exactly one *interface*, which handles actual communication.

3.2.1 Component

A component is a piece of designed hardware. It has multiple ports over which communication can take place, i.e. date is sent from or to the component, esp. the control ports *clock* and *reset*. Usually components receive data, process it and send back some results, though possibly on another port.

Components have to be described in VHDL for embedded system design with the Xilinx toolsuite. Such VHDL definitions can be referenced in the board description language provided by the HOPP Driver Generator. They are parsed and translated into corresponding software components with ports as defined in VHDL.

3.2.2 Port

Ports are used to send data to or receive data from components. A port is always assigned to a single component, but a component can have multiple ports. Ports can be receiving ports, sending ports or dual ports. They can have an arbitrary bitwidth. Since the bitwidth of the microblaze as well as the arm processor is limited to 32-bit, the port class has to split greater bitwidths in several 32-bit packages, but hide this process from the user. The encoding format, i.e. order of bits, also is hidden from the user.

3.2.3 Communication Interface

The communication interface is hidden from the user. Each component has exactly one such interface and a communication interface has to be attached to a component. The communication interface wraps all communication between host software and board software. The interface abstracts from the physical network interface and provides a homogeneous api for components and ports. Network interface specific initialization is generated as well and is not required by the user (other than annotating configuration details in the board description).

Note, that a board is also only allowed to use a single interface. Consequently, components using the same interface are physically located on the same board.

Currently, three communication interfaces are envisioned:

- **Ethernet**, which is already implemented
- **USB/UART**, which is considered as second interface and
- **PCIE**, which will not be implemented in the initial driver generator.

Ethernet Communication over Ethernet is based on the lightweight IP stack, originally developed by Adam Dunkels[2].

3.3 Generation Backend

This section is used to introduce the C generation backend. A reference to the third Katja report might be useful.

3.3.1 CModel

The c model provides data types representing a c/c++ program. Note, that the model is neither complete nor always valid, i.e. not all c programs can be described using this model and it is possible to specify a model not translating into valid c. Still, the model simplifies the process of code generation. The model is used for generating C as well as C++ code.

```
MFile ( String name, MDefinitions defs, MStructs structs, MEnums
    enums, MAttributes attributes, MMethods methods, MClasses
    classes )

MClass ( MModifiers modifiers, String name, MTypes extend,
         MStructs structs, MEnums enums, MAttributes attributes,
             MMethods methods, MClasses nested )

MModifier = PRIVATE() | PUBLIC() | CONSTANT() | STATIC() | INLINE()
// this is a comment
```

First of all, each file has a name. A file consists of several definitions, structures, enums, attributes and methods. Files also can contain several classes. These classes again have a name and can contain all these components including other classes. In addition, classes can contain modifiers and inherit components from other classes. The allowed modifiers are private, public, constant, static, and inline. Note, that not all of these modifiers are class modifiers, and several combinations of modifiers are invalid (e.g. private and public). The model relies on the developer to choose modifiers according to the modified program part.

```
MDefinition ( String name, String value)
MStruct ( MModifiers modifiers, String name, MAttributes attributes
    )
MEnum   ( MModifiers modifiers, String name, Strings values )
```

Definitions, structs and enums mark rather trivial tuple productions. A definition simply assigns a name to a value. Enums list a number of possible values.

```
MAttribute ( MModifiers modifiers, MAnyType type, String name,
    MCodeFragment initial )
MCodeFragment ( String part, MIncludes needed )
```

[2]For documentation, a wiki exists for the lwip stack at http://lwip.wikia.com/wiki/LwIP_Wiki

Attributes are similar to definitions, but are typed and may also be left unassigned, using an empty code fragment. The `MIncludes` is required if the type of the attribute is not defined within this c file itself.

```
MMethod ( MModifiers modifiers , MReturnType returnType , String name
    , MParameters parameter , MCode body )

MReturnType = MAnyType | MVoid() | MNone()

MParameter ( MParamType refType , MAnyType type , String name )
MParamType  = VALUE() | REFERENCE() | CONSTREF()

MCode ( Strings lines , MIncludes needed )
```

Methods have a return type and a list of parameters. The method body is also more complex than a simple code fragment and can consist of several lines, which are not checked any further in this model. The return type can be any c type as well as void. For constructors in c++, the return type `MNone` is used. Parameters also have a type and name. Furthermore, the mode of parameter passing has to be specified.

```
MAnyType = MType              ( String name )
         | MArrayType         ( MAnyType type , Integer length )
         | MPointerType       ( MAnyType type )
         | MConstPointerType  ( MAnyType type )
```

The type system of the model supports arrays as well as (const) pointers. The basic type is the `MType`, which consists only of a string that has to reference an existing c type, e.g. "int" or "struct student". This type can then be extended using pointer or array types. So `MArrayType(MType("int"), 5)` would mark an integer array of length 5. Note, that these types are nested semantically rather than in the same order as in C. Consequently, a point type of a const pointer type of type integer will be translated into `int const ** a`.

3.3.2 Unparser

Different unparsers are used, to translate the C model in actual code. For each instance of the model, a header file and a corresponding source file, either C or C++, has to be generated. These unparsers are called depending on the particular instance of the model. Files intended to be loaded to the board have to be plain C, files intended for the client side can also be C++ files. The unparsing itself is realized using the visitor pattern. Each visit method appends code to a string buffer depending on the visited element.

Header Unparser The header unparser is used for both, unparsing C as well as C++ code. Consequently, it doesn't filter any constructs, but accepts everything specifiable with the model.

This unparser generates only signatures for all methods and only the declarations of attributes and enums. However, the header file will contain all includes referenced within the model.

Plain C Unparser Since plain C doesn't have any concept of classes, using a model with classes in this unparser will result in exceptions. Otherwise, all components and combinations are accepted. The source file will only import the corresponding header file, no other headers or sourcefiles.

C++ Unparser For the C++ unparser, like with the header unparser, every component and combination is allowed. The source file will only import the corresponding header file, no other headers or sourcefiles.

3.4 Transformation

Explain the transformation of the source file into the c model.

Bibliography

[1] Fischer, T.: Data Binding for Schemata with Integrity Constraints and Atomic Procedures - A Generative Approach for Object-Oriented Languages. Master's thesis, University of Kaiserslautern (2012)