

HOPP Driver Generator Documentation

Thomas Fischer

March 15, 2013

Contents

1	Introduction	3
1.1	Origins and Goals	3
1.2	Terminology	4
2	Getting Started	5
2.1	Setup	5
2.2	Board Description Language	6
2.3	Process	6
2.3.1	Building the Generator	6
2.3.2	Running the Generator	6
2.3.3	Executing your Application	8
3	Driver Description	9
3.1	API	10
3.1.1	Component	10
3.1.2	Port	11
3.2	Architecture	11
3.2.1	Queueing	11
3.2.2	I/O Threads	12
3.2.3	Bitwidth Translation	13
3.3	Protocol	14
3.3.1	Control Flow	14
3.3.2	Message Encoding	17
3.3.3	Sequence charts	20
3.4	Current Driver Implementations	20
3.4.1	C++ Host-Side Driver	20
3.4.2	Virtex 6 ML 605 Board-Side Driver	22
4	Generator	29
4.1	Used Libraries	29
4.1.1	JFlex & CUP	29
4.1.2	Katja	29
4.2	Generation Process	30
4.3	CModel	30

4.4	Unparser	32
-----	--------------------	----

Chapter 1

Introduction

This document is intended to give an overview over the HOPP Driver Generator for both, users and developers. While the first chapters are more addressed towards users of the generator, the later ones are more addressed towards developers.

The HOPP Driver Generator itself is intended to provide embedded system developers with a simple, convenient interface to communicate with their designed hardware components. To our knowledge, such an interface for communication with VHDL components has not been done so far, though extensive research exists concerning efficient communication between PCs and FPGAs over different communication channels, mainly Ethernet [3, 1, 2].

Our interface has been prototyped with a C++ frontend communicating with a Xilinx Virtex 6 ML-605 FPGA over Ethernet. The design enables easy extension to support other frontend languages, FPGA boards or transport media.

1.1 Origins and Goals

The HOPP Driver Generator originally was and currently is developed by the Software Technology Group of the University of Kaiserslautern in cooperation with the Microelectronic Design Research Group of the University of Kaiserslautern.

The design goals of this project are:

- Probably better filled out by the EIT department ;)
- An easy-to-use driver and driver generator
- Well-documented api
- Modularity and extensibility
- Meaningful error description

1.2 Terminology

This section explains the terminology used in this document and the project in general.

Driver The complete software product is called the *driver*. It enables **software-side** communication with the hardware platform.

Board / board-side The driver is split in two parts, one of which has to be uploaded and executed to 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.

Host / host-side In contrast to the board-side driver, the host-side driver is the part of the driver which 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.

Chapter 2

Getting Started

The purpose of this chapter is to explain how to properly build and use the generator. This part contains all steps required in order to run the generator and acquire drivers for the specified board.

2.1 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. This requires (for example) Windows users to adjust their `PATH` variable.

Java The generator is implemented in Java due to tool support and the environment of the Software Technology Group. Consequently, a JDK version 6 or above is required.

Gradle Gradle¹ is a build tool, similar to *Maven* or *Ant* (like *Make*, but - for the most part - easier to manage within larger projects). It supports the dependency management from Maven while retaining the flexibility of Ant. Plugins required by the build process are automatically downloaded by Gradle.

Mercurial Mercurial² is a distributed versioning tool, comparable with *GIT*, *Bazaar* or (to some degree) *Subversion*. The sources of the driver generator are located in a mercurial repository. If you acquired the sources (and this document) through other means mercurial is not required.

Doxygen Doxygen³ is used for generation of a Java API-like html description of the driver API. While this generation is not required for the driver, it is highly recommended for easier integration of the generated driver.

¹available at <http://www.gradle.org/>

²available at <http://mercurial.selenic.com/>

³available at <http://www.doxygen.org/>

C/C++ Compiler Since the host side driver is written in C++, it is also required to have a C/C++ compiler present.

Xilinx Toolsuite In order to generate an `.elf` file that can be used to program the FPGA, the Xilinx toolsuite is required. This includes ISE for generating IP Cores out of VHDL files, XPS for composing these and EDK for actually generating the file for the defined hardware platform.

We try to avoid to require the user to actually design anything in the Xilinx suite, but only use it for synthesising the board hard- and software. Both, `.mhs` file and all source files for the EDK, are created by the driver generator.

It is furthermore desirable to skip user-interaction with the EDK completely. This would require generation of the board support package and external call of Xilinx' compiler.

2.2 Board Description Language

In this section, the board description language is introduced. This language is used to specify a board, for which a driver should be generated.

2.3 Process

If the tools described in Section 2.1 are correctly installed, the following steps should provide you with a working version of the driver generator.

2.3.1 Building the Generator

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.⁴

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 a jar package is `gradle jar`. Simply type it in a shell in the project root repository, where the file `build.gradle` is located. Running the build file will generate an executable jar package under the path `<project>/build/libs`. Navigate there and try running it using the command `java -jar <name of the jar package>`.⁵

2.3.2 Running the Generator

The HOPP Driver Generator can be called using the command line interface (CLI). The CLI offers several parameters to further configure the run of the generator, listed in Table 2.1.

⁴The repository is located at <https://softech.informatik.uni-kl.de/hg/ag/hopp/>. Please note, that authorization is required.

⁵This should result in the usage help and an error, since no `.mhs` file has been specified.

-s	--server	Specifies the server backend. This specifies, for which target platform the driver should be generated. The default setting is a Virtex 6 ML 605 FPGA (currently, no other options are present).
	--serverDir	Specifies the destination of the server directory. All files for the server (i.e. files used to generate the board side driver) will be generated into the specified directory. If none is specified, a directory "server" will be generated inside the current working directory.
-c	--client	Specifies the client backend. This specifies, in which language the client software should be generated. The default setting is a C++ frontend (currently, no other options are present).
	--clientDir	Specifies the destination of the client directory. All client for the server will be generated into the specified directory. If none is specified, a directory "client" will be generated inside the current working directory.
-v	--verbose	The driver generator will print out additional console output. This makes it easier to track errors in the driver generator or the used source file.
-d	--debug	The driver will be generated with additional debug console output. This makes it easier to track errors in larger test cases.
-h	--help	Lists all CLI parameters and a short explanation. The generator will abort after parsing this parameter and not generate anything.
	--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 - together with the debug parameter. It may be used for actually debugging the generator then ...).

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 the following line:

```
java -jar driverGenerator.jar [OPTIONS] <.mhs file>.
```

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.

In addition to the required sources, documentation for both host- and board-side sources is generated, using doxygen.

2.3.3 Executing your Application

Now the regular design process from Xilinx can be continued. The next step would be writing a host application that uses the generated host-side driver and API. After programming your FPGA with the `.elf` file generated using the board-side driver, your program should be able to communicate with VHDL components on the FPGA through this API.

Note, that it is required to call the `setup` method before actually sending data over the driver. It is also recommended, to shut the driver threads down properly by using the provided shutdown method. Both these methods are **/will be** contained within the components file.

Chapter 3

Driver Description

This section should provide a conceptual overview of the driver parts. For pure users of the driver generator, only parts of the client side are really relevant. Developers might also be interested in the server part.

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 specification using doxygen (see Chapter 2 for details on how to enable documentation generation).

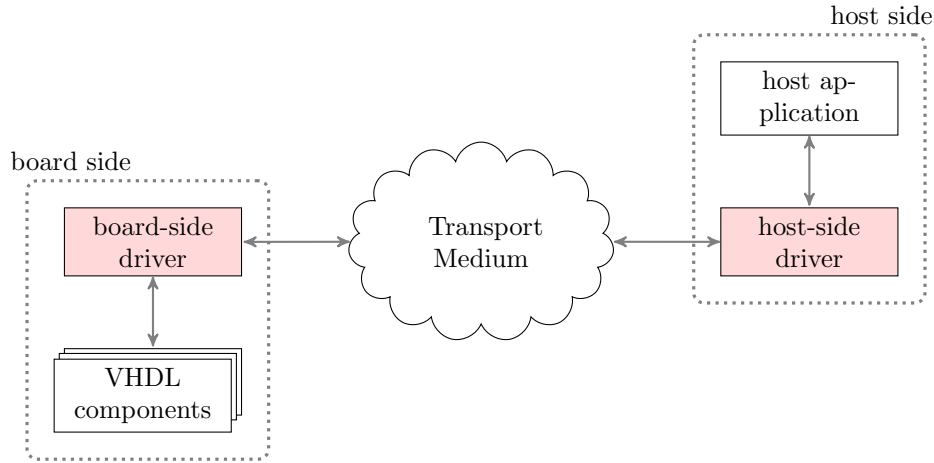


Figure 3.1: A high-level view of the data flow from a host application to vhdl components on the board through the generated driver

The overall architecture of the driver is depicted by Figure 3.1. 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 VHDL components on the FPGA. Results are sent back through the same chain.

3.1 API

This section describes, how the host-side API, the hardware designer programs his application against, should look like. This helps designers in writing their application and describes how host-side drivers in other languages should be implemented in general.

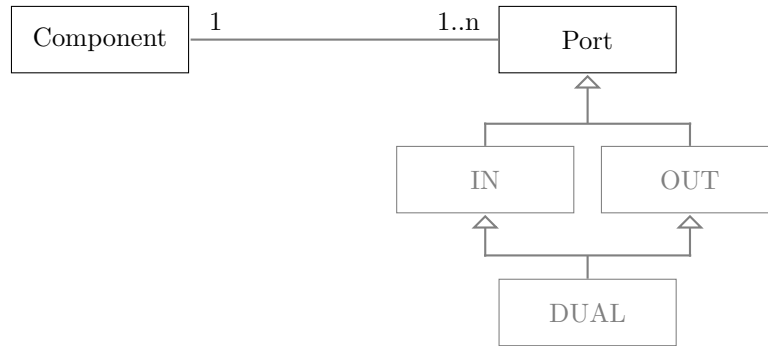


Figure 3.2: Exposed classes of the host part

The exposed architecture of the host software is depicted in Figure 3.2. A board is described using multiple *components*. A component is a designed hardware unit, which can have several *ports*, over which data can be sent to or from the component.

3.1.1 Component

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

The host driver contains an abstract generic component, describing components in general. For each user-defined core, a new subclass is created, which contains the ports specified in the core description. For each instance of a core on the board, an object of the cores subclass together with all its ports is instantiated in the driver respectively. Communication with the boards components happens through these port objects.

Details about the implementation of components can be found in Section 3.4.1.

GPIO components These components are specialised, predefined I/O components, used to directly input or output a signal on the board. The GPIO representation in the driver enables the host-side application to write or read this signal. A GPIO component is either used as input or output component. An input component, enables input of a signal. An example for such a component is the pushbutton component on the Virtex 6. An output component

does the opposite and displays a signal on the board. The Virtex 6 has an LED component which is classified as output component. The signal to be displayed can be written to such a GPIO component.

Since such components only hold a single state and perform no processing, communication is not handled via ports. Instead, it is possible to directly read or write the state of the component.

3.1.2 Port

A port marks an AXI stream interface 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 sending (*in-going*), receiving (*out-going*) or bi-directional (*dual*). Values sent to or received from a port have an arbitrary, but fixed bitwidth. Ports are the only way to communicate with components or the board in general (beside aforementioned GPIO components).

Ports allow *synchronous* as well as *asynchronous* communication. A synchronous write to a port waits for the message to be delivered to the component. A synchronous read waits for a value to be received. Asynchronous operations do not wait, but return immediately. The values are expected to be written or read asynchronously. While the order between messages to a single port is maintained, the order between messages to different ports may differ from the order of performed operations.

Details about the implementation of ports can be found in Section 3.4.1.

3.2 Architecture

3.2.1 Queueing

Due to restricted resources on the board and desired parallelism between the different components and ports, queueing is an important aspect of the driver and different kinds of queues are introduced.

The first and most important queue is the client-side *write queue*. Such a queue is introduced for each in-going port. Writing to a port does not directly result in a message sent to the board, since the board has restricted resources and may not be able to process or even store the value. The client queue provides a first mechanism of *flow control*. Values are queued up at the client, when the board buffers have been filled and no more values can be received, until the board signals that it can receive more values now. Additionally, the client buffer offers *congestion avoidance* to a small degree, since several small values can be packaged together in one message. This results in a smaller amount of messages being sent over the medium.

The client also contains *read queues* for each out-going port. These queues cache values received by the board for read operations performed by the application. Values are pushed from the board into these queues automatically. These queues lead to smaller resource consumption on the board, since only a

small number of out-going values have to be cached. Read cycles are faster, since no polling messages have to be sent but available values can directly be consumed. On the downside, client-side read queues lead to some possibly unnecessary traffic on the medium, since values are sent without being requested at the board. **The worst case here being a component that constantly generates values, that usually are only required by request, for example a random number generator. We therefore should also allow the usual poll semantic in specification language, drivers and protocol. These do not require a read queue, but write results directly in the provided memory area.**

With only client-queues, the board-side driver would only be capable of receiving and processing a single value at a time, which has to be directly written to a component, resulting in an increase of messages and therefore communication overhead. In order to enable reception of multiple values at once, a *board-side queue* (or *software queue*) is introduced for each in-going port on the board as well. Received values are stored in these queues first and forwarded to the target component later on, if the component can process further values. In contrast to the client-side queues, are size-restricted and can only hold a certain number of values, specified in the board description file. Similar queues are introduced for out-going ports, to reduce traffic caused by values being sent back to the host-side driver. These queues cache results from components which are then sent in one message.

Since the FPGA usually only has a single processor and is not capable of multi-threading, only a single board-side queue can be read at a time, and only a single component can receive values. In order to feed all components with faster with values, a queue is implemented in hardware and placed in between the component and the stream interface of the processor. These *hardware queues* provide the component with values as soon as it requires a new value and therefore remove the idle time between the end of a computation of a component and the board-side driver thread serving the component new values. Since these queues are actually less efficient in terms of resource consumptions on the board than the software queues, they are even smaller in size. Again, similar queues on out-going ports cache results, so the component does not have to wait for the driver thread to read the value from the port before starting the next computation.

3.2.2 I/O Threads

As stated in Section 3.1.2, the driver should support asynchronous writes and reads. The order of messages to the same port has to be maintained, while the order of messages to different ports can be changed. The processing of operations at ports is independent from other ports. In particular, a synchronous operation on a port should not influence an asynchronous operation on another port.

Consider the following example to understand why asynchronous operations should still be processed during a synchronous operation. A board specification

contains an adder component with two in-going ports and an out-going port. If a values are written at both ports, they will be consumed and a result will be returned at the out-going port. Consider further, that a value is currently stored at port A. The user now writes another value to port A asynchronously, followed by writing three values to port B synchronously. The value for port A cannot be written directly, since port A is still blocked with a value. However, the first value for port B can be written, and both values are consumed. Now the second value for port B can be written, blocking the port for the third value. This third value cannot be written, until the second value is consumed, which requires another value at port A. If write operations are performed asynchronously, the asynchronous write to port A will never reach this port, since the application is still blocked from the write to port B. Allowing asynchronous writes to be processed in parallel resolves this situation. The value for port A can be written in between the writes to port B or at any point afterwards.

Asynchronous operations can be realised by dedicated I/O threads, handling communication between host-side and board-side driver. These threads modify the client-side queues described above in Section 3.2.1. In general, a single thread, that covers the complete communication is be sufficient. For better separation between writing and reading operations, two threads can be introduced for, one for each of these operations. This solution if preferable, since reading from the medium usually involves longer periods of listening for messages, which would unnecessarily delay write operations. A third solution, implementing the independence of operations on different ports, is introduction of a single thread for each port. This, however, results in a possibly large number of threads. Depending on the host platform, a smaller number is preferable.

The threads have to be started with a special method `startup()` before using the API and should be shut down afterwards with another method `shutdown()`.

3.2.3 Bitwidth Translation

As described in Section 3.1.2, ports allow arbitrary bitwidth. As the size of the AXI stream interface directly connected to the FPGAs processor is usually fixed, translation to this fixed size has to occur somewhere in between. Performing this translation on-board consumes limited board resources. Doing the translation on the client however increases communication overhead. Depending on the application, both solutions can make sense.

A board-side translation requires send messages to be augmented with the bit-size of each value. Values can than be padded and/or split, according to the required and the actual bitwidth. This translation is individual for each port.

A client-side translation does not require the bit-size to be transmitted. Padding and translation in this case is done immediately before transmission. This reduces the required memory for client-side queueing compared to direct padding before the client-side queue. Re-translation to the actual bitwidth has to be done directly after the AXI stream interface of the processor.

3.3 Protocol

This section covers the transmission protocol between client and server application.

The protocol is based on the assumptions made above about client and server, specifically the used buffers and client I/O threads allowing parallelism with sending and receiving messages. Furthermore, the underlying medium and protocol are expected to be reliable and order-preserving.

3.3.1 Control Flow

This section describes the state of the driver and summarises messages that are sent during each transition. To simplify the control flow graphs, the following variables and functions are introduced:

- **v** Values of some sort (e.g. an array of integers).
- **a** Memory addresses (e.g. an array of addresses).
- **n** The size of the board-side queue (statically known).
- **i** An unsigned (i.e. positive) integer number, smaller than **n**.
- **size(q)** The number of values currently held by queue **q**.
- **empty(q)** **true**, if **size(q) == 0**, **false** otherwise.
- **full(q)** **true**, if the queue is full, **false** otherwise.
- **store(q,v)** Stores the values **v** into queue **q**.
- **take(q)** Returns the first value of a queue **q**. Removes the value from the queue.
- **drop(q,i)** Drops the first **i** values of **q**.
- **peek(q,i)** Obtain the first **i** values of **q**. Returns a smaller number of values, if only less are available, i.e., **size(q) < i**.
- **asgn(a, v)** Assigns a value **v** to an address **a**.

Messages are exchanged between the application and the client-side driver, as well as between client-side and board-side driver. The following messages are the most important ones influencing the state of the driver:

- **wrt(v)** Request from the application to write values **v** to the port.
- **read(a)** Request from the application to read values to the addresses **a**.
- **notify** Notifies the application that all tasks queued up at a port have been processed.
- **data(v)** A data message containing values **v** for or from a port.
- **ack(i)** Acknowledges successful reception of **i** values.
- **poll** A request message for additional data.

Host-Side Driver

Since values can be written asynchronously, each port maintains its own state and can perform transitions independent of the other ports. The state of the host-side driver is constructed from the individual states of all ports.

The state of an in-going port is physically represented by two variables \mathbf{q} and \mathbf{s} . \mathbf{q} denotes the queue, while \mathbf{s} stores the number of values in transit, i.e. those, that have been sent but not yet acknowledged. A more abstract view on the state of an in-going port is provided in Section 3.3.1. Each state represents a combination of these variables. Changes to these variables are not explicitly denoted in the diagram.

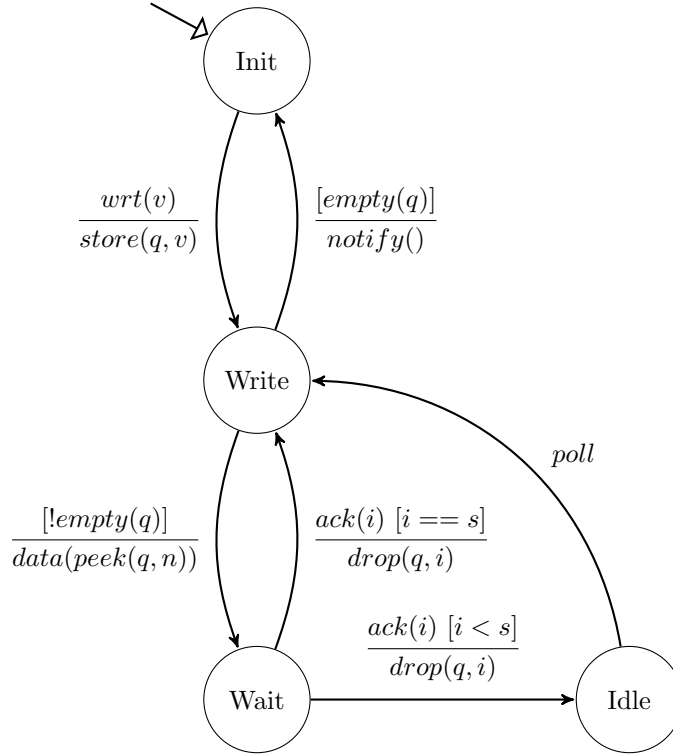


Figure 3.3: Host-side control flow graph of an in-going port

Several loops have been left out in order to simplify the graph. Messages **ack** or **poll** in other states than specified in the graph will simply be ignored. An application write in any state will result in the values to be appended to \mathbf{q} .

The state of an out-going port is represented by two variables \mathbf{q} and \mathbf{r} . \mathbf{q} is a queue of memory addresses, where values should be read to, \mathbf{r} is a queue of values read from the medium but not from the application so far.

Values are automatically pushed forward from the board and stored in \mathbf{r} . If

the application requests a **read**, values are shifted from **r** into **q**. The application is notified, once all read requests have been served.

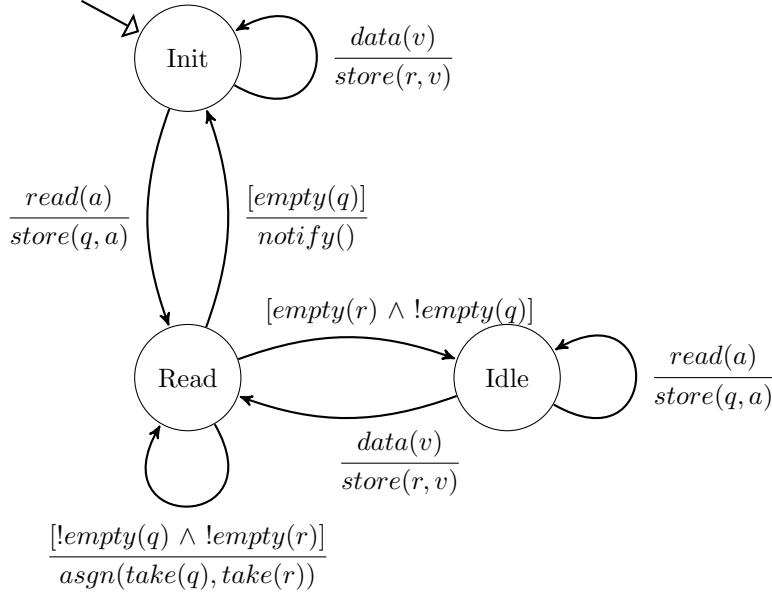


Figure 3.4: Host-side control flow graph of an out-going port

Board-side Driver

The state of the board-side driver is also represented as concatenation of the states of all ports. The state of a port on the board is represented by the corresponding software queue **q** and the corresponding hardware queue **r**.

Since only a single thread can run on the FPGA, a scheduler determines which port can make a step. The default scheduler switches to a different port, once the current port cannot perform any more steps. It is possible for the user to define a more elaborate scheduler, if so desired.

An in-going port, as shown in Section 3.3.1, sends acknowledgements for received data packages and stores received values. If it has received values and can write to the hardware queue at some point, it switches to consuming received messages. Messages are then shifted from the software queue to the hardware queue, until either the hardware queue is filled or the software queue is emptied. If the software queue was full before shifting the first message, a poll is sent in addition. After shifting all values possible, the port switches back to listening for more values.

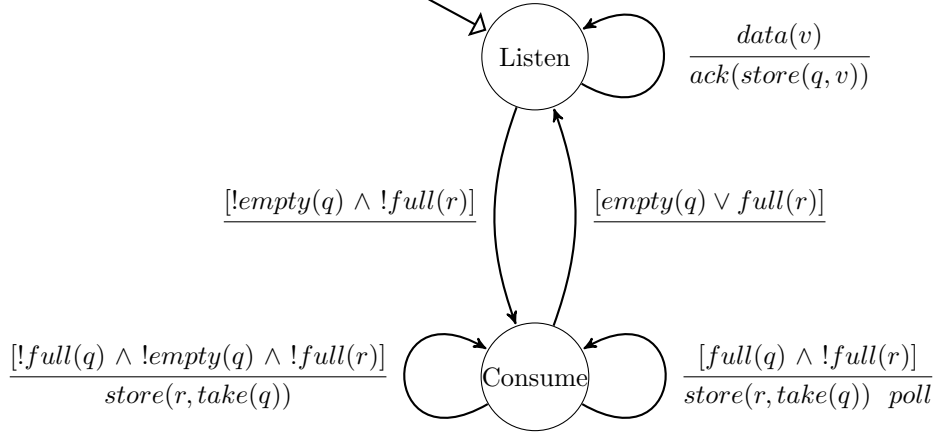


Figure 3.5: Board-side control flow graph of an in-going port

The out-going port is even simpler (see Section 3.3.1). It shifts values from the hardware queue to the software queue as long as possible. Then it sends a data message with all values contained in the software queue.

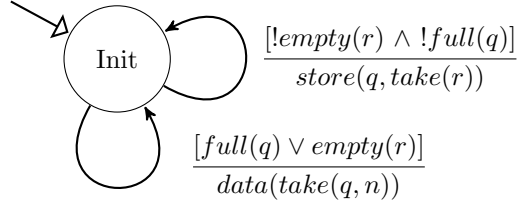


Figure 3.6: Board-side control flow graph of an out-going port

3.3.2 Message Encoding

This section covers the translation of the above messages into messages on the communication medium. Messages can be split into header and payload. The header describes the payload to follow, the payload contains a number of 32-bit values that are sent to or from a component.

The first 8 bit of the header are reserved for the *protocol version*, the message was encoded with. The only version currently available is version 1.

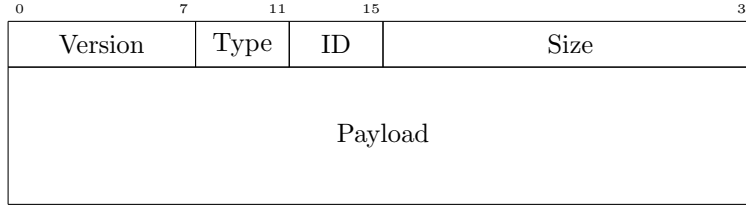


Figure 3.7: Bitorder of the message on the medium

Procotol Version 1 This version reserves the next 4 bit of the header for the type of the message. In general, a message looks like depicted in Figure 3.7. Values in this protocol version are expected to be properly aligned with 32-bit, meaning that, as described in Section 3.2.3, values are padded host-side to a multiple of 32-bit and then split into 32-bit blocks.

The control flow graphs in Section 3.3.1 specify what types of messages are required. A short overview of the messages together with their type encoding and their fields is provided in Table 3.1. The following paragraphs provide a more detailed description of the message and their meaning.

Message	Type	ID	Size	Payload
Data	1001	Port ID	Payload Size	Yes
GPIO	1110	GPIO ID	GPIO State	No
Ack	1111	Port ID	Ack Count	No
Poll	1010	Port ID	Unused	No
Reset	0000	Unused	Unused	No
Debug	0111	Debug Type	Payload Size	Yes

Table 3.1: Overview of messages in protocol version 1

Data Message A data message marks either a set of values for a specific component being sent from the host to the board or a set of values from a specific component being sent from the board to the host. As such, it requires an identifier for the target or source component as well as the size of the contained payload in words (i.e. 32-bit values). Values sent with a data message are expected to always be full 32-bit values.

Acknowledgement The acknowledgement confirms reception of a number of values by a specific component. For this purpose, no payload is required. Instead, the number of acknowledged values is encoded within the *size* field.

Data Request (Poll) A data request is used to inform the host, that additional values can now be received. This is necessary if the queue was full

beforehand, i.e. a partial acknowledgement was (most likely) sent. The poll does require neither a payload nor a size, but only the identifier of the component, that can now receive values.

GPIO Message A GPIO message is a special type of data message, addressed to a GPIO component. GPIO components use their own address space, disjunct from the addresses used by "normal" components. They are not connected via AXI Stream interfaces but direct memory addresses, consequently they do not influence the port restriction on the microblaze.

Furthermore, GPIO components do only store their current state and perform no calculation like vhd components. The state is represented by an 8 bit value and is encoded directly in the size field instead of the payload. Storing only the current state also means, that no queues exist for GPIO components and no acknowledgements are required. The new state is simply written into (or read from) memory

Reset Message This message is not specified in the protocol above. A reset message sent by the host-side driver resets the state of the board-side driver, clearing all queues and setting the reset flag for all components. The board-side driver acknowledges a successful reset by answering with a reset message. The target and size fields are unused by the reset message.

Debug Message This message also is not specified in the protocol. It marks a notification of some sort, sent by the board. This can be debug output of the driver running on the microblaze, a warning message about skipped messages or an unhandled error, that occurred on the board. For these messages, the payload contains a string. The target bits are used to differentiate between the debug message type. The bit encoding is shown in Table 3.2. Values in between have been left unused for future use (e.g. finer grained warnings or info messages).

Severity	Encoding
Info	0011
Warning	1000
Error	1101

Table 3.2: Debug message encoding

While it is possible to provide debug output over the JTag cable, the FPGA is programmed with, this quickly slows down computation with larger debug outputs. Using the Ethernet connection also for debug output vastly accelerates such computations.

3.3.3 Sequence charts

Some sequence charts to visualise the envisioned process. For example, Figure 3.14 shows in detail, how a non-blocking write is handled. First of all, the values are stored client-side. The application receives a pointer to a `write` object, which represents the current state of the scheduled write, i.e., how many values have been received by the component on the server.

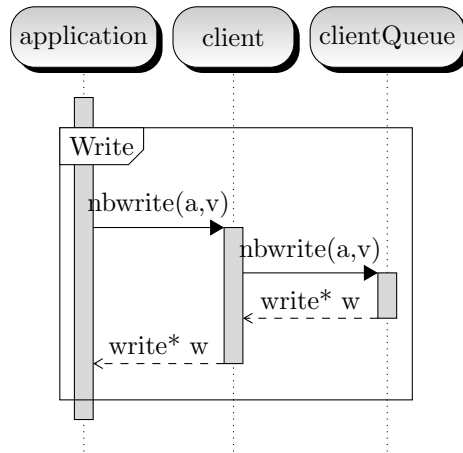


Figure 3.8: An application writing values through the client-side driver api

3.4 Current Driver Implementations

3.4.1 C++ Host-Side Driver

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

The actual communication is performed in the *communication medium*, which is unique per driver and does not require additional interaction from the user.

Communication medium

Each driver is bound to a single communication medium, which wraps lower-level communication (essentially transport layer and below) between host and board driver. The communication medium abstracts from the actually used technology and provides a homogeneous api for the I/O threads. Network interface specific initialisation is generated as well and is not required by the user (other than annotating configuration details in the board description).

Currently, three communication mediums are envisioned:

- **Ethernet Lite**, which is already implemented

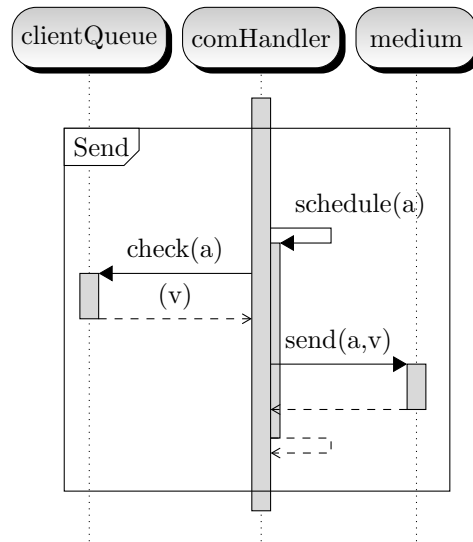


Figure 3.9: A client sending data

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

Ethernet Lite Communication over Ethernet is based on the lightweight IP stack, originally developed by Adam Dunkels¹.

It is possible to implement more efficient Ethernet communication using dedicated VHDL components instead of running the lwip stack on the general purpose CPU. Such a component could easily be integrated in form of a new Communication Interface. An example for a dedicated VHDL communication component together with a client API is described in [1, 2].

Describe implementation of general interface methods in Ethernet as well as challenges and problems this interface poses.

USB/UART While communication over USB/UART is rather slow, it provides a more simple method of communication than using Ethernet.

PCI Express According to [1], there exists a Xilinx wrapper for PCIE communication.

¹The lwip stack is documented with a wiki available at http://lwip.wikia.com/wiki/LwIP_Wiki

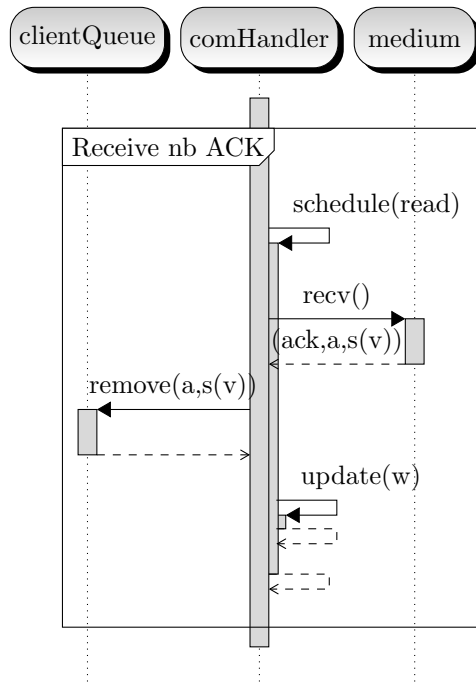


Figure 3.10: The client-side driver receiving an acknowledgement from the server

3.4.2 Virtex 6 ML 605 Board-Side Driver

The board-side driver consists of several files that can roughly be split into three groups: files handling communication with the host-side driver, files handling communication with board components and setup files.

All these 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.

Restrictions

This section covers restrictions of the Virtex 6 board and (if available) implemented workarounds.

Port Count The microblaze on the Virtex 6 only allows 16 AXI master interfaces and 16 AXI slave interfaces. As a result, only 16 in- and out-going ports can be specified when generating a Virtex 6 board driver. Circumventing this restriction is possible by implementation of a multiplexer which delegates values to one of several components, but this is left to the user.

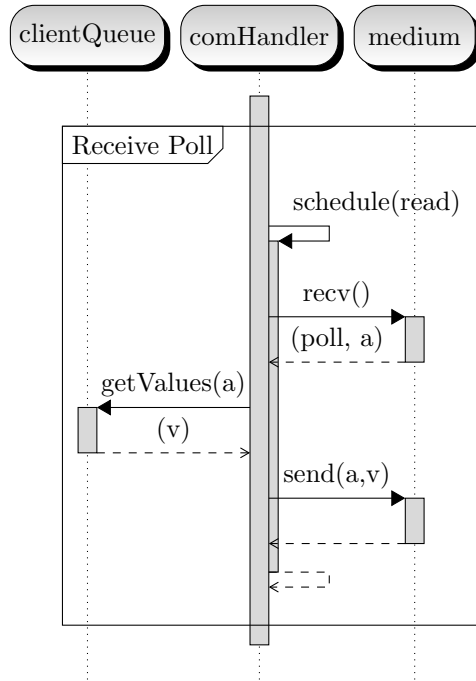


Figure 3.11: The client-side driver receiving a data request from the server

Bitwidth Translation As explained in Section 3.2 and Section 3.3, transmitted values are padded to a multiple of 32-bit for transmission and have to be re-translated at the board-side driver. Since AXI stream ports of the microblaze processor on the Virtex 6 (as well as the ARM) are fixed at 32-bit, this translation has to occur **after** the software part of the board-side driver. Consequently, a bit-translator component is put in-between the hardware queues on the board and the interface to the microblaze. This translator is omitted, if the values expected by the attached component are indeed 32-bit values, in order to save some board resources.

Implementation Details

Medium The medium folder contains a source and header file for the medium the board is attached to. In this file, medium-specific setup is performed and listening to incoming messages is handled. An incoming message is passed to the protocol interpreter, which distributes the content of the message in accordance to its header.

Components The components folder contains code for component-specific setup and communication between the boards general purpose cpu and designed

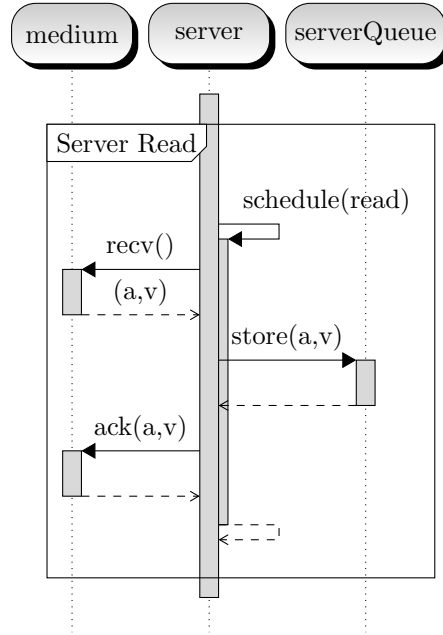


Figure 3.12: The server-side driver receiving a data package from the client

IP cores. Code for controlling gpio components will also be generated in this folder.

General Several files are generated directly in the `src` folder of the project. These files contain data types used by all file groups, files required for initialisation, and files linking between medium and components.

Among these files is the `main.c`, which calls the initialisation procedures of the medium and all components and starts the scheduling loop. This loop performs three kinds of operations:

First, it checks for incoming messages and process their contents. Most of the time, this includes storing values in the in-going microblaze queue and acknowledging them. More details about message handling can be found in Section 3.3.

The loop also shifts messages from in-going microblaze queues to in-going hardware queues and vice versa from out-going hardware queues to the out-going microblaze queue. Finally, once the out-going queue is filled or no more values are available, it warps values into a message and write this message to the medium.

The loop can be overridden by the user, but is required to perform all these operations at some point for the driver to work correctly.

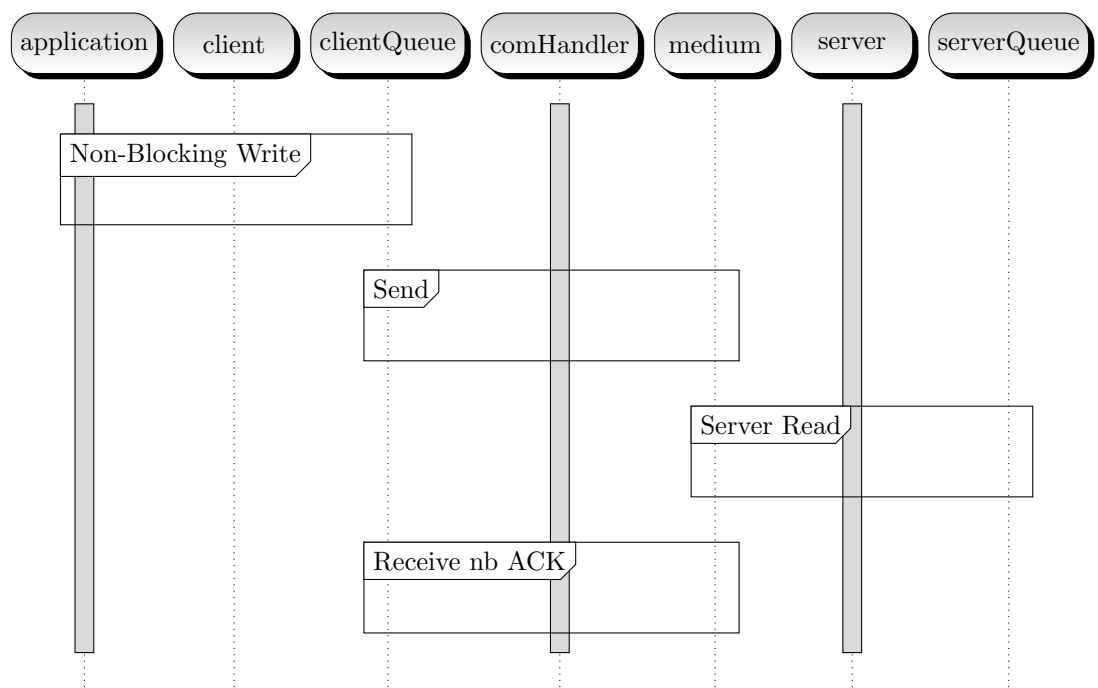


Figure 3.13: Composed example of a non-blocking write using the building blocks defined before.

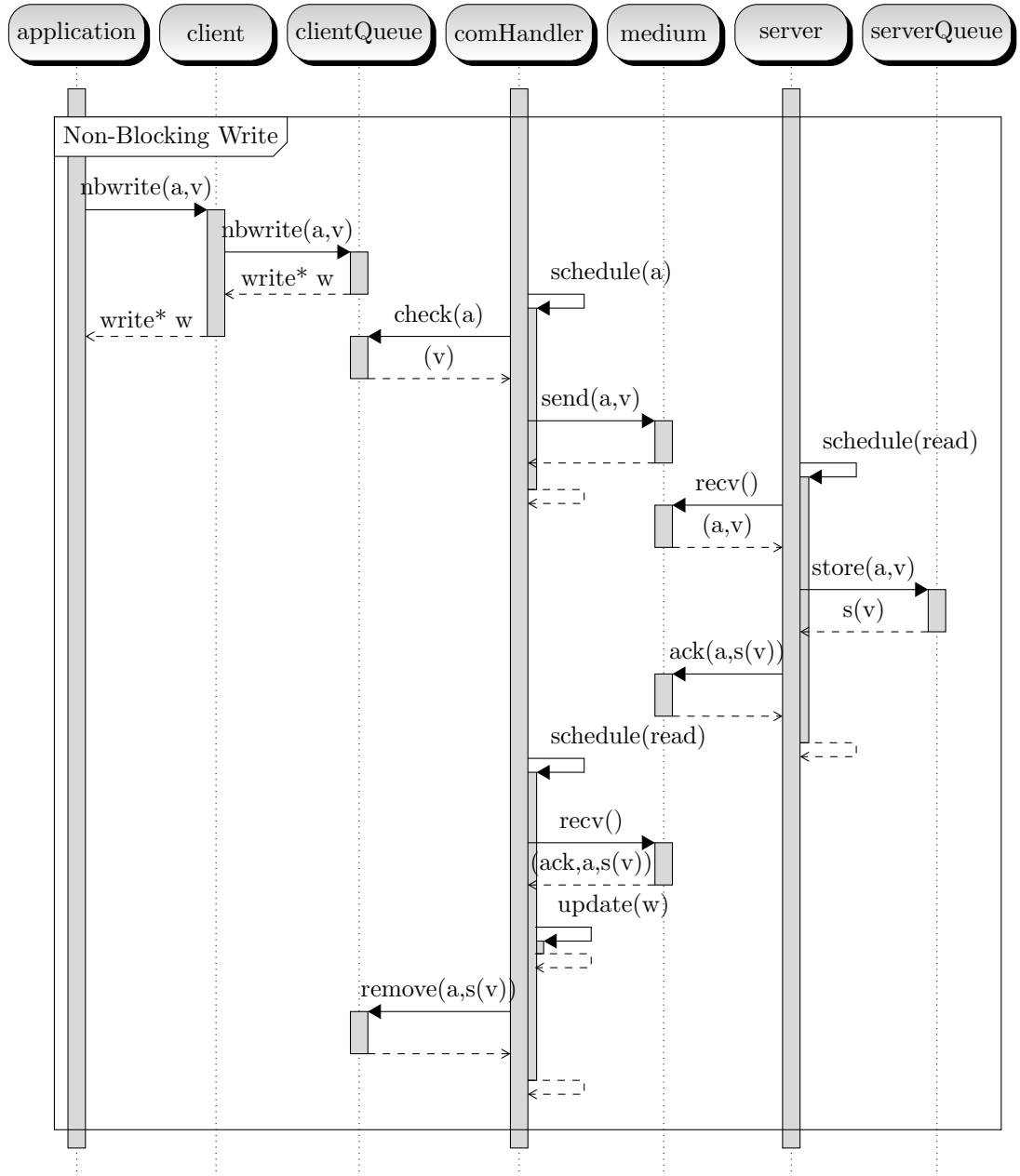


Figure 3.14: A sequence diagram for a non-blocking write. Note, that non-blocking only means, the client does not wait for the target **component** to receive the message, but still waits for the client queue to receive the values.

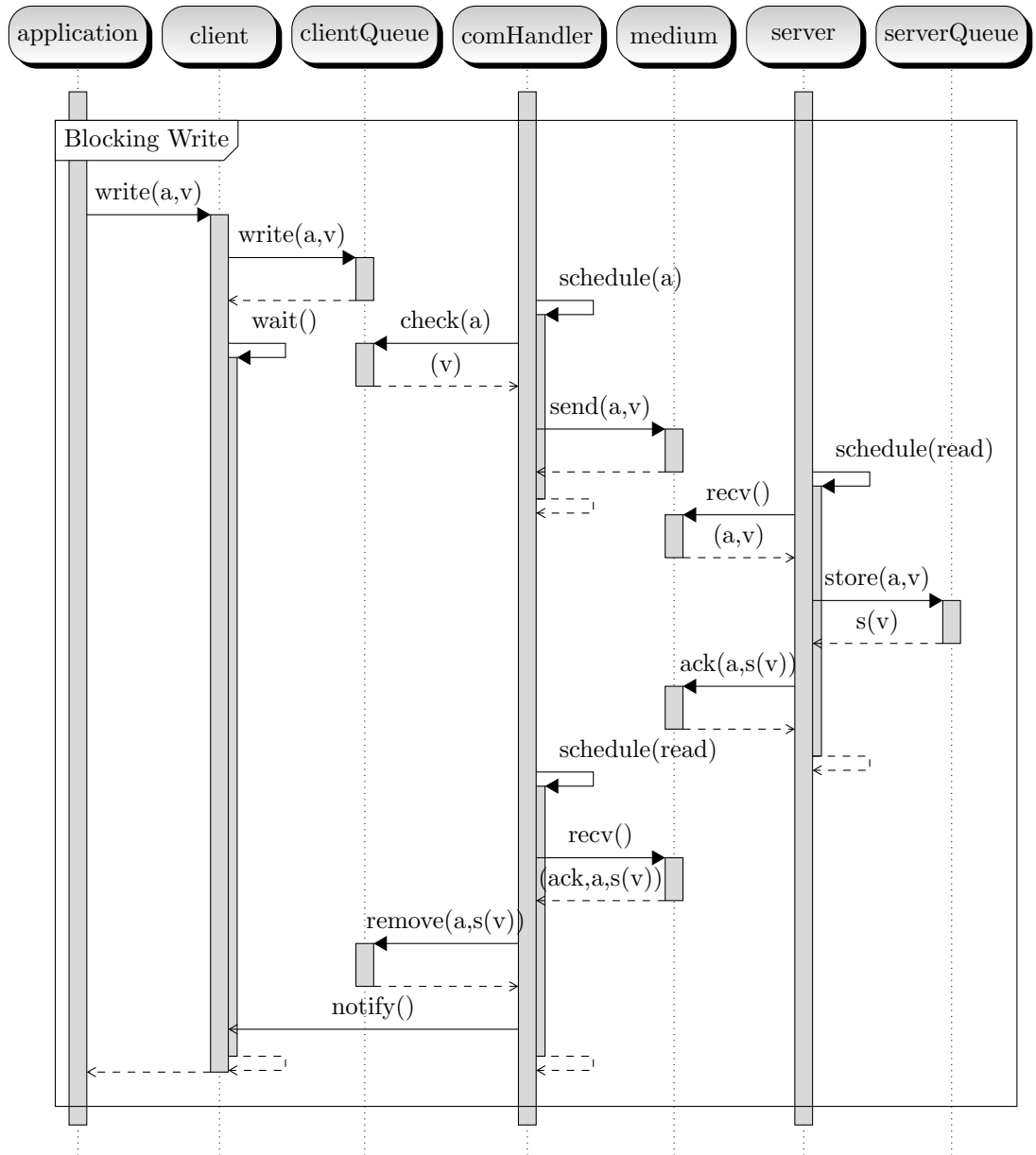


Figure 3.15: A sequence diagram for a blocking write. Note, that only the application is blocked, not the communication handler, which may process non-blocking writes sent before the blocking write **after** the non-blocking write.

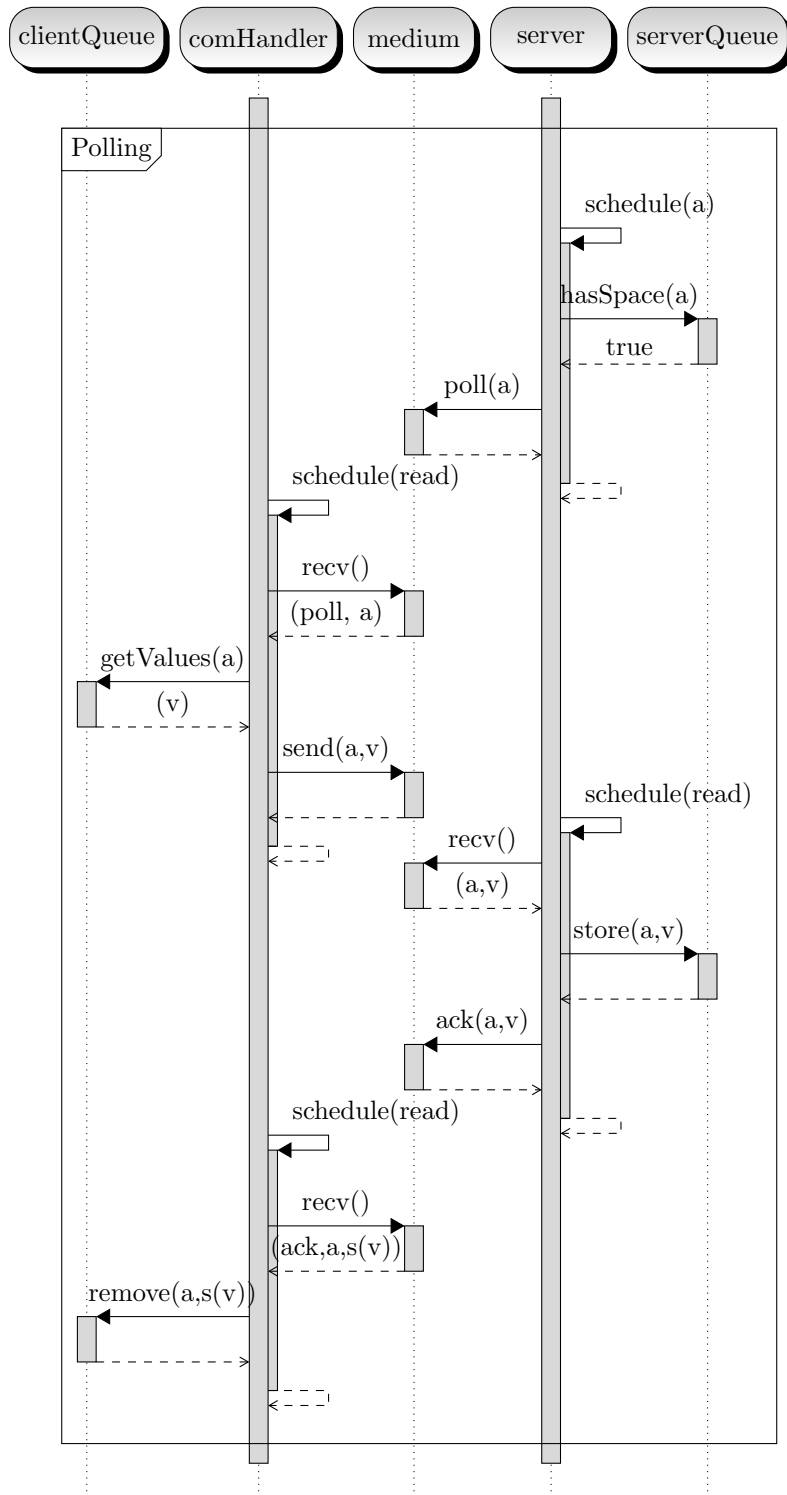


Figure 3.16: A polling scenario. The server scheduler reaches component a and registers free space in the queue (that was formerly full). He then polls values for component a. The poll triggers sending of data at the client, which is later received in a read phase of the server.

Chapter 4

Generator

This chapter is used to explain the code generator itself. This information is intended for future developers of the driver generator. [A reference to the third Katja report might be useful.](#)

4.1 Used Libraries

This section gives a short overview over the used libraries and explains what for and why they are used. These libraries are included in the repository and build jar file and require no further user interaction. Still, as the generator code depends on them, they are introduced here for future developers.

4.1.1 JFlex & CUP

[This can be kept short with a reference to the JFlex/Cup documentation](#) Flex is a scanner generator, CUP a parser generator. Both together are currently used to parse `.mhs` files into abstract syntax trees. [In future versions, they will be used to parse the DSL of the driver generator.](#)

4.1.2 Katja

The driver generator uses the Katja tool, developed by the Software Technology Group of the University of Kaiserslautern. This tool generates several data types¹. 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 4.3.

¹Since 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://softtech.informatik.uni-kl.de/Homepage/Katja>

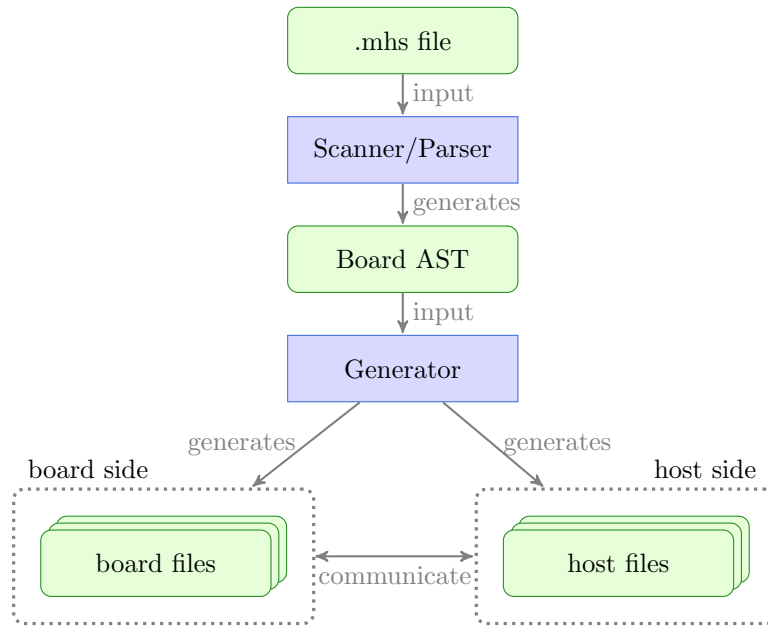


Figure 4.1: A rough sketch of the translation process so far

4.2 Generation Process

The overall process of the driver generator is described by Figure 4.1. 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 models of all required header- and source files. These models are translated into files by the C/C++ unparsers.

Several backends exist to create different types of models. Host backends generate models for source code running on a host machine, board backends generate models for source code running on a board respectively. These backends implement a `Visitor`, which visits all components of the board, and manipulate the source model accordingly. Currently, the only available backends are the C++ host backend and the Virtex 6 ML 605 board backend.

4.3 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 ( MDocumentation doc, String name, MDefinitions defs,
        MStructs structs, MEnums enums, MAttributes attributes,
        MMethods methods, MClasses classes )

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

MModifier = PRIVATE() | PUBLIC() | CONSTANT() | STATIC() | INLINE()

```

First of all, files can be documented using an **MDocumentation** element. If the documentation is not empty, a **@file** tag is attached to indicate this as a file documentation for doxygen. A file has a name and 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. Classes also can be documented. 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 ( MDocumentation doc, String name, String value)
MStruct ( MDocumentation doc, MModifiers modifiers, String name,
          MAttributes attributes )
MEnum ( MDocumentation doc, 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. All three can be documented.

```

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

```

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. They also can be documented.

```

MMethod ( MDocumentation doc, 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`, `struct student` or `enum day`. 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`.

```

MDocumentation ( Strings doc, MTags tags )

MTag = PARAM      ( String name, Strings details )
      | RETURN     ( Strings details )
      | THROWS     ( String type, Strings details )
      | DEPRECATED ( Strings details )
      | SEE        ( String see )
      | AUTHOR     ( String name )
      | SINCE      ( String date )

```

For generation of the API, Javadoc style documentation has to be added to the model. A `MDocumentation` element contains documentation for the following block as well as possibly several tags. Tags can be parameter or return value descriptions of a method/procedure, descriptions for exception behaviour, deprecation descriptions or references to other elements of the code. The `JAVADOC_AUTOBRIEF` option is set in the generated doxygen config files, resulting in the first sentence (concluded by a dot and following space or newline) will be used as short description for the documented program part. Since neither the driver generator nor doxygen perform sanity checks, we rely on the user to only introduce meaningful tags for a documentation element. Note further, that the order of tags influences the order of elements in the generated API description.

4.4 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.

Bibliography

- [1] Alachiotis, N., Berger, S.A., Stamatakis, A.: Efficient PC-FPGA Communication over Gigabit Ethernet. In: Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology. pp. 1727–1734. CIT '10, IEEE Computer Society, Washington, DC, USA (2010), <http://dx.doi.org/10.1109/CIT.2010.302>
- [2] Alachiotis, N., Berger, S.A., Stamatakis, A.: A Versatile UDP/IP based PC-FPGA Communication Platform. In: ReConFig 2012 (2012)
- [3] Lofgren, A., Lodesten, L., Sjöholm, S., Hansson, H.: An analysis of fpga-based udp/ip stack parallelism for embedded ethernet connectivity. In: NORCHIP Conference, 2005. 23rd. pp. 94 – 97 (nov 2005)