Hochschule
Augsburg University of
Applied Sciences

The *ASTERICS* Book

User and Reference Guide for the
Augsburg Sophisticated Toolbox for Embedded Realtime Image Crunching Systems (ASTERICS)

Michael Schäferling, Julian Sarcher, Alexander Zöllner,
Philip Manke and Gundolf Kiefer

With contributions from:
Michael Zink

Hochschule Augsburg – University of Applied Sciences
Efficient Embedded Systems Group

`http://ees.hs-augsburg.de`

Version 0.0.3
February 21, 2020

Document History

Version	Date	Description
0.0.1	2017-08-15	Set up document format
0.0.2	2019-06-19	New release alongside a new public version of ASTERICS
0.0.3	2019-11-07	Clean up Makefile and fix title page (full version)

About this Document

This document is an attempt to collect all relevant information about the *ASTERICS* framework - concepts, usage, development and reference guides - handy in one place.

It is a ”living document”, which is maintained and developed by the *ASTERICS* community in a self-organized way. Its quality relies on the contributions of all those many developers contributing to the framework. The *ASTERICS* Book is distributed ”as is” without any warranty regarding correctness or completeness of information.

Everybody using *ASTERICS* is welcome to help improving this book. Please send your contributions or suggestions for improvement to:

Philip Manke	<philip.manke@hs-augsburg.de>
Michael Schäferling	<michael.schäferling@hs-augsburg.de>
Gundolf Kiefer	<gundolf.kiefer@hs-augsburg.de>

Contents

Part I.

User and Developer Guide

1. Overview

by Gundolf Kiefer, Michael Schäferling, Alexander Zöllner

1.1. Background

As FPGAs are growing in their capacities, hardware structures of increasing complexity can be implemented. This allows to map more sophisticated algorithms to these structures, enabling ever more complex applications. Due to the wide range of algorithms and their varying requirements on resources and architecture, utilizing heterogeneous platforms is becoming increasingly popular. These allow to utilize powerful hardware/software codesigns, by utilizing dedicated hardware structures along with a complex software stack, as well as a full-featured operating system. Modern systems-on-chips (SoCs) even allow to integrate these codesigns on a single chip.

Computer Vision (CV) has been a significant area of research for quite some time and has already benefited various fields of industry, such as detecting faulty parts built by automatized production lines. Advances in technology within recent years also enabled using CV for embedded applications, such as advanced driver assistance systems, which use CV for line and traffic sign recognition. These tasks are also indispensable for autonomously driven cars. Another application is validating the status of a construction site's progress, such as determining the number and location of pipework installation within ship and plant constructions. Here, CV is used for tasks such as detecting geometric primitives and translating them to the actual objects for a CAD model.

However, CV is still a computational and resource demanding task, which is expected to no longer be automatically solved by more powerful processors becoming available due to reaching technological limits in foreseeable future. Rather, innovative and efficient structures have to be developed to further advance the fields of application using CV. The *ASTERICS* framework already offers dedicated hardware structures for speeding up CV tasks by simultaneously using only little resources. Exemplary structures include an implementation of a Hough Transform variant for detecting arbitrary objects and lines [**kiefer˙configurable˙2016**] and a pipeline-based architecture for applying 2D window filters on images [**pohl˙efficient˙2014**], with more hardware structures being continuously added.

In order to seamlessly integrate *ASTERICS* into CV applications and reducing development time, a full-featured software stack is also provided, which can be integrated into an operating system. Further, integration into the Robot Operating System (ROS) is offered and integration into the commonly used OpenCV library is actively being worked on.

The main goal of *ASTERICS* is speeding up and enhancing CV applications in a convenient and efficient manner by utilizing its hardware structures along with its software stack.

1.2. What is *ASTERICS*

1.2.1. Overview

ASTERICS is an open toolbox for building hardware accelerated image processing chains [**EES˙ew˙2015˙asterics**]. Therefor, hardware designs for an increasing range of algorithms are offered, which are developed in the form of self-contained modules. This allows to combine them to arbitrary image processing chains in a modular manner, in order to meet the requirements of a certain application. *ASTERICS* also contains a number of software drivers and utilities for interfacing its hardware modules. In order to cover a wide range tasks, *ASTERICS* covers all categories for image processing, namely *point operations*, *window filter operations*, *semi-global/patch-based operations* and *global operations* [**pohl˙efficient˙2014**]. Figure 1.1 shows an exemplary image processing chain, where critical aspects of the image processing task is sped-up in hardware. The results are passed to software using main memory for further processing or may be displayed on a monitor.

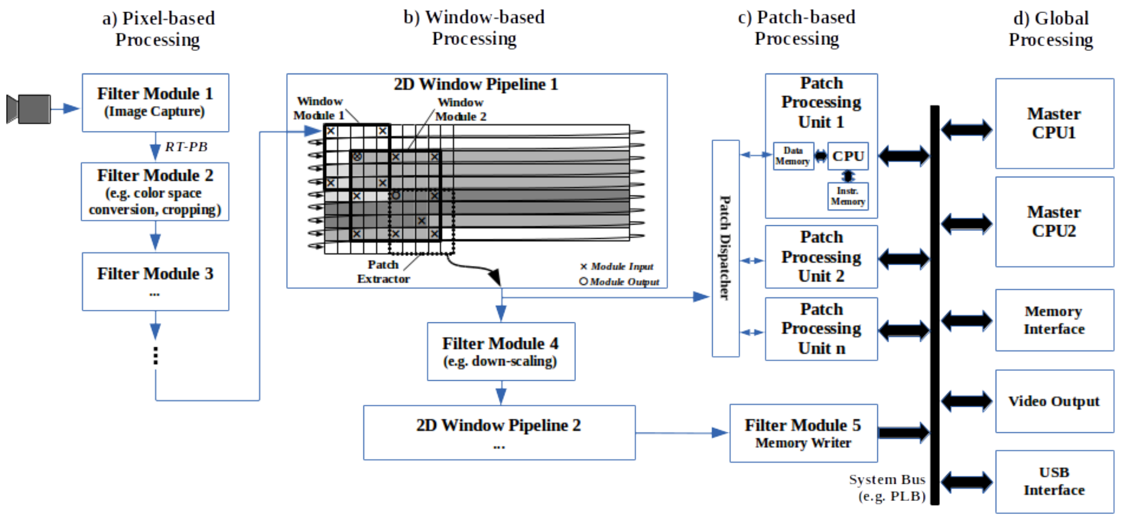

Figure 1.1.: Example *ASTERICS* system

1.2.2. Interfaces

ASTERICS utilizes a modular principle for combining its modules to arbitrary image processing chains. Great effort has been made to create a set of interfaces for defining the inter-module communication and hardware-software interaction to guarantee a certain behavior within the processing chain and to prevent data loss. The interfaces of *ASTERICS* are organized as following:

- General Interfaces (Chapter 5.1)
- Common Per-Module Signals (Chapter 5.2)
- The *ASTERICS* Streaming Interface (`as_stream`) (Chapter 5.3)
- The *ASTERICS* 2D Window Filter Interface (Chapter 5.4)

1.2.2.1. General Interfaces

This type of interfaces covers general concepts for interaction between hardware and software for a single module as well as a processing chain as whole. Here, the software is accessing the hardware for obtaining status information and influencing the behavior of the hardware. This includes the types of errors which may occur within the hardware which have to be addressed by software and the expected behavior of the hardware in case the software *resets* single modules or the processing chain as a whole. Additional topics are the version management and the i2c bus interface for communicating with external hardware components, which are not part of the programmable logic.

1.2.2.2. The Common Per-Module Signals

These interfaces are used for hardware internal communication to request status information and trigger certain operations, which affect modules as a whole. These status information include the `READY` signal, which indicates whether the hardware module is currently operable or the `SYNC_ERROR` which signals that data has been lost at some point during operation. Regarding the operations affecting modules as a whole, signals such as `RESET` and `FLUSH` may be present for the module.

The aforementioned interfaces can either be utilized by other *ASTERICS* hardware module or by some logic for controlling the entire image processing chain, e.g. to reset all modules at once.

1.2.2.3. The *ASTERICS* Streaming Interface (`as_stream`)

Signals which are required for controlling the actual data transfers between hardware modules, e.g. the inter-module communication, are summarized here. The signals `DATA` and `STROBE` are used for setting up the most basic data transfers. The `as_stream` interface defines additional signals, mainly for synchronization to provide more detailed information about the data layout, e.g. `HSYNC` for indicating the start of a new line. Additionally, the `STALL` signal is defined by the `as_stream` interface for requesting to temporarily suspend data transfers, in case data is processed at a varying pace across hardware modules.

1.2.2.4. The *ASTERICS* 2D Window Filter Interface

This type of interface defines a set of signals for hardware modules which operate on a 2D sliding window buffer.

1.2.3. Basic Modules

ASTERICS offers a wide range of hardware modules for building image processing chains. Modules who fulfill a less complex or common task are categorized as basic modules within *ASTERICS* . The memory modules for transferring data between hardware and software (`as_memreader/-writer`), converters (`as_invert`) or adapters (`as_disperse/as_mux`) fall into this category. Although there is no set rule or checklist for defining *basic modules* within *ASTERICS* , modules belonging to this category are usually not particularly relevant for the image processing task itself but rather serve a supporting role. For this reason, multiple instances of modules of this type can be commonly found across an image processing chain.

The currently available basic modules of *ASTERICS* are presented in detail in chapter 7. Some modules may not yet be publicly available due to being currently in development or being in the staging process to be released. In case you cannot find the documentation for a specific module, please contact one of the authors. Depending on the stage of the module, its documentation (if already available) or further information regarding the module will be provided.

1.2.4. Sophisticated Modules

Contrary to basic modules, sophisticated modules serve a distinct image processing task which is rather complex and requires a dedicated hardware design in order to be performed efficiently regarding resource consumption and processing speed. Modules belonging to this category are for example the *Universal Hough Transform* (`as_uht`), the *Non-Linear Image Transformation* (`as_nitra`) or the *Canny Edge Detector* (`as_edge_and_scale`). Due to their unique design and efficient implementation, there is usually also one or more scientific publications associated with the module.

A detailed description of the sophisticated modules of *ASTERICS* are presented in chapter 8. Some modules may not yet be publicly available due to being currently in development or being in the staging process to be released. In case you cannot find the documentation for a specific module, please contact on of the authors. Depending on the stage of the module, its documentation (if already available) or further information regarding the module will be provided.

1.2.5. Tools

A tool is a piece of software, which aids its user at a regularly required or tedious task regarding building or using an image processing chain as a whole or certain aspects of it. Tools are usually script-based and perform their task in a (semi-)automatic manner. An exemplary tool is the *2D Pipeline Generator* for building 2D-sliding window buffers along with required interfaces for accessing the data.

A detailed documentation of the available tools of *ASTERICS* are provided in chapter 6.

1.2.6. Software Stack and Options

The software stack is an accumulation of software drivers for the various modules of *ASTERICS* , abstraction layers from vendor and platform dependencies and definitions of the actual image processing chain as well as the environment of the software. The contents of the software stack allows to operate any *ASTERICS* -based image processing chain in a convenient manner bare-metal and applications with operating system support.

The contents of the software stack and data transfer schemes are outlined in chapter 4.

1.3. Organization of this Document

This document is divided in two parts. The first part is a user guide and contains all material necessary to understand the basic concepts and to get started using *ASTERICS* or eventually developing new *ASTERICS* modules or tools.

- Chapter 1 gives an overview on the *ASTERICS* project as a whole.
- Chapter 2 contains all information necessary to start using *ASTERICS* and to develop systems containing *ASTERICS* chains.
- Chapter 3 contains all information required to contribute to the *ASTERICS* project by developing new modules or tools.

The second part serves as a reference guide.

- Chapter 4 describes the organization of the highly configurable and portable software stack together with all aspects related to the hardware-software interfaces.
- Chapters 5, 6, and 7 refer to the three main dimensions of *ASTERICS* and give a reference on the interfaces, tools, and commonly used modules, respectively.
- Chapter 8 gives an overview of the more complex modules of *ASTERICS* , such as feature detection or object recognition.
- Chapter 9 lists a number of ”ready-to-use” systems, deploying a specific *ASTERICS* chain.

In summary, if you are...

- ... new to *ASTERICS* and want to learn about its capabilities and get the demos running on your computer, you should read Chapter 2.
- ... a new member or partner of the *Efficient Embedded Systems (EES)* group or for some other reason plan to work on the *ASTERICS* framework, you should read Chapter 3 to get acquainted with the code organization of the project.
- ... already an experienced *ASTERICS* developer, you will certainly always remember that Chapters 4 through 9 serve as a reference manual where you can find any information you may ever need. However, the evolution of this document itself is part of the *ASTERICS* development process. If you come across anything missing or outdated, you will also certainly feel the responsibility to add or correct the missing information.

1.4. Further Reading

Although the present document covers most information required for utilizing and developing for *ASTERICS* , there are also a number of additional resources available. These resources mainly address more general topics regarding *ASTERICS* .

- The ASTERICS *homepage* of the *EES* group introduces the framework and lists recent work.
- The *ASTERICS* wiki page gives a detailed overview of the framework’s structure. The source code can also be obtained from here.
- The source code of the *ASTERICS* modules (hardware and software) is documented by *Doxygen*.

- The various concepts and ideas revolving around *ASTERICS* are presented in a series of *publications*. These comprise, among others, a configurable architecture for the *Generalized Hough Transform* [**kiefer˙configurable˙2016**], a pipelined architecture for *feature detection* [**pohl˙efficient˙2014**] as well as a module for removing distortion and rectifying images [**pohl˙efficient˙2012**].

1.5. Terminology and Conventions

1.5.1. Terminology

An *ASTERICS module* is an image processing module, which may be implemented in hardware, in software, or in a combination of both. Modules primarily implemented in hardware (e. g. filter modules) are referred to as *hardware modules*, those primarily implemented in software (e. g. `as_memio`) are referred to as *software modules*.

An *ASTERICS chain* is a complete sub-system of connected *ASTERICS* modules, typically implementing one or even multiple complete image processing chains. At system level, an *ASTERICS chain* is represented by an IP core on the hardware side and by an *ASTERICS Support Package* in the software side.

1.5.2. Conventions

Throughout this book, the following conventions are used:

- Signal, variable of function names that can also be found in the code, are written in a `typewriter font`. Hardware signals are written in `UPPER_CASE`, C code functions and module names in `lower_case`.
- *Italic font* is used for emphasis.
- **Bold font** is used for definitions.

2. Using *ASTERICS*

by Michael Schäferling, Gundolf Kiefer, Philip Manke

2.1. Installation

The installation process currently just comprises cloning the *ASTERICS* GIT repository (Chapter 3) or extracting a repository snapshot. There are no further installation steps required, but this is likely to change in the near future (due to the fact that there's a lot of ongoing development, especially in the field of the *ASTERICS* tools).

2.2. Getting Started with the Supplied Demo Systems

The *ASTERICS* framework provides demonstration systems to get in touch with the concepts of the framework and its modules. They also may be used as a basis for further development.

Demo systems are equipped with a `Make` based build system. With this, a specific *ASTERICS* chain is prepared, which then is integrated into the system during the build step. Also, within the build step the needed software for proper operation is also supplied. Finally, the demo systems can be put into operation on the specific board.

At the moment, only the system contained in `"systems/as_refdesign_zynq/"` is up to date and fully supported. We strongly recommend using this system to evaluate *ASTERICS* and to use this as a basis to develop your own system. It was tested using Vivado versions 2017.2, 2018.3 and 2019.1. The other systems were built using older Vivado versions and older versions of the *ASTERICS* modules and may no longer be functional.

For more information, read the `README` file in the root directory of the respective demo system and section 2.2.1.

2.2.1. Automated Build Process

In section 6.2.1 a short user guide describes the necessary steps for building the demo system `as_refdesign_zynq` with a focus on the *ASTERICS* system generator *Automatics*. Here, *Automatics* is described in less detail with more focus on the build process itself. The following prerequisites are necessary to build and use the demo system:

- Linux based operating system (Microsoft Windows and MAC OS have not been tested).
- Python 3.5 or higher. Necessary to run *Automatics*, the *ASTERICS* system generator.

- An *ASTERICS* installation.
- Optional: A Xilinx Vivado and XSDK installation. This is needed if you want to evaluate the system on a supported board. Note that the automated process is currently only available for the Xilinx toolchain.
- Optional: A hardware target. To run the demo system as is, the Zybo[1] board and an OmniVision OV7670 camera module are required. Necessary cable drivers and board files need to be installed. For other hardware targets, the constraint files and the Vivado project (block design) need to be modified manually.
- Optional: To view the demo system in action, a monitor with a VGA or HDMI input connected to your Hardware is required.

  NOTE: If your Linux distribution does not support the `source` command by default (Ubuntu, Debian, ...), it does not use *bash* as the standard shell. You need to either modify this setting system-wide or explicitely run each of the files that is passed to the `source` command in the steps below.

Follow these steps to build the demo system `"as_refdesign_zynq/"`, located in 'asterics/systems/':

1. Get the latest snapshot of the GIT repository from:
   `http://ees.hs-augsburg.de/asterics/`
2. Extract the snapshot to your preferred installation directory and move into the directory containing the `asterics` folder using your preferred console application.
3. Set up a workspace directory and move into it:
   ```
   > mkdir asterics-workspace && cd asterics-workspace
   ```
4. Copy the directory `asterics/systems/as_refdesign_zynq/` to your workspace:
   ```
   > cp -a ../asterics/systems/as_refdesign_zynq/ .
   ```
5. Optional: If you intent to build the Vivado IP-Core (also needed to build the whole system), you need to source your Vivado installation:
   ```
   > source <VIVADO-DIR>/settings64.sh
   ```
6. Source *ASTERICS* by running:
   ```
   > source ../asterics/settings.sh
   ```
7. Move into `as_refdesign_zynq/` in your workspace:
   ```
   > cd as_refdesign_zynq
   ```
8. Now you can build the system. Different targets are available, depending on the tools at your disposal:

[1] `https://store.digilentinc.com/zybo-zynq-7000-arm-fpga-soc-trainer-board/`

a) You can generate only the ASTERICS source files using:

```
> make asterics_core
```

b) If Vivado is available, you can generate the *ASTERICS* IP-Core (to be used in a Vivado project) using:

```
> make asterics_vivado_cores
```

c) You can build the entire FPGA project (after the Vivado IP-Core was generated, see above), using:

```
> make build_system
```

Note that this will take some time, depending on the capabilities of your development system.

d) If you have a ZyboBoard and OV7670 image sensor board handy, you can program the hardware and software onto the target boards FPGA by running:

```
> make run
```

e) Alternative to b) - d): To build, implement and compile the hardware and software and flash it onto the board in one step, you can execute:

```
> make build_and_run
```

or

```
> make all
```

2.3. How to Design *ASTERICS* Systems

The *ASTERICS* generator tool *Automatics* automates most of the process of generating and building an ASTERICS IP-Core. Automatics requires Python 3.5 or newer installed on your system. To use it, you first need to source *ASTERICS* by running:
`source <asterics>/settings.sh`
Note that Automatics can automatically run Vivado to package the *ASTERICS* IP-Core. To do this, Vivado has to be installed and sourced *before* sourcing *ASTERICS* , using:
`source <Vivado install location>/Xilinx/Vivado/<version>/settings64.sh`

Next you should set up a workspace where you want to build the system. Copy the file `user_script.py` from `<asterics>/tools/as-automatics/` into your workspace. This file contains all the information needed by the generator to build a very basic *ASTERICS* system. The system description uses Python syntax, however, you do not need knowledge of Python syntax to work with the script. Just by copy-pasting statements and altering parts of code lines, even complex *ASTERICS* systems can be assembled. The file `demo_script.py` contains further examples of *ASTERICS* system descriptions using Automatics. Furthermore, the scripts contain code comments describing the functionality of each command and additional commands and capabilities of Automatics.

To start the generator, run:
`python3 user_script.py`

For details on Automatics and its capabilities, read section 6.2.

The process of manually creating an *ASTERICS* system consists of several steps:

- Design and implement an *ASTERICS* IP core which contains the desired image processing operations by including and connecting the needed *ASTERICS* modules. This task can be done manually by editing the following design units of the *ASTERICS* IP core:
  - **asterics.vhd** is the top level file of the *ASTERICS* IP core. It instantiates modules for master and slave bus access and the as_main design module (which contains the actual image processing chain).
  - **as_main.vhd** implements the image processing chain by instantiation and connection of the appropriate *ASTERICS* modules.
- Integrate the *ASTERICS* IP core into a system-on-chip (e.g. by establishing bus connections and mapping external signals for an image sensor module).
- Develop and integrate code which configures and instructs the modules of the *ASTERICS* IP core. This may be bare-metal software or software running on Linux which interfaces with the hardware modules using an *ASTERICS* device driver.

2.4. The *ASTERICS* Source Tree

The *ASTERICS* source tree consists of the following elements:

- tools: Supporting tools (e.g. as-automatics)
- support: *ASTERICS* support files (including base for *ASTERICS* software support package 'ASP')
- modules: Image processing modules (both hardware and software)
- ipcores: IP cores which are part of the *ASTERICS* project
- systems: Small ready-to-use systems with preconfigured core structures

3. Developing *ASTERICS*

by Michael Schäferling, Michel Zink, Alexander Zöllner, Gundolf Kiefer

3.1. Organization of the GIT repositories

The *ASTERICS* Frameworks code base is organized in a set of GIT repositories.
Note: The GIT repositories mentioned here are only accessible from **within the local network** of the University of Applied Sciences Augsburg. Visit the website http://ees.hs-augsburg.de/asterics for access to a snapshot of the repository.

The *ASTERICS* GIT repository contains a subset of publicly accessible modules, systems and tools and is organized according to the source tree structure (see section 2). You can view the codebase online by browsing

```
$ https://ti-srv.informatik.hs-augsburg.de/gitweb/?p=asterics.git
```

or get it by

```
$ git clone https://ti-srv.informatik.hs-augsburg.de/repo/asterics.git
```

3.2. License and Copyright

This document and the mentioned download link refer to the free and publicly available part of ASTERICS. This part is generally licensed under the LPGL (see the LICENSE file in the root folder of the repository for details).

However, there are more ASTERICS modules available, which are presently not published under an open source license, but that can be made available individually on a per-project basis. These include modules for:

- the Generalized Hough Transform and other variants of the Hough Transform [**kiefer˙configurable˙2016**], [**hough˙object˙recog**]
- low latency lense distortion removal and stereo rectification (NITRA) [**nitra˙paper0**], [**nitra˙paper1**]
- efficient, on-the-fly point feature extraction (SURF algorithm) [**pohl˙efficient˙2014**]

Please contact the Efficient Embedded Systems (EES) group at the University of Applied Sciences Augsburg (see "authors" section on top of the file) for further information on using these modules and collaborating with the EES group.

If you add something to the project that is licensed under a different license, please append to the LICENSE file. Make sure that the license you are using is compatible with the LGPL.

Acknowlegements:
Parts of this work have been supported by the German Federal Ministry for Economic Affairs and Energy (BMWi), grant number ZF4102001KM5 (2015-2018).

Parts of this work have been supported by the German Federal Ministry of Education and Research (BMBF), grant number 17N3709 (2009-2013).

3.3. Coding Conventions

3.3.1. C Code

by Michel Zink

3.3.1.1. Introduction

This chapter defines rules and requirements for maintaining and developing C source code for the *ASTERICS* framework. The main emphasis hereby lies on the style and coding conventions of C source code itself, files- and directory-naming included.

The purpose of this section is to enable good comprehensibility of the developed C source code across various software developers and increase its general quality. Thus, reducing the intial time required for acquiring understanding of a given source code and simplify its maintenance as well as accelerate the integration of new newly developed functionalities into the *ASTERICS* framework.

The given requirements outlined in this document are based on the C99 standard as common ground.

This chapter uses the expressions *must/has to*, *strongly recommended*, *should* and *can* to give information about the relevance of the coding convention in question. Rules or sections marked as *must* are binding and are to be applied without exception. *Strongly recommended* parts are to be always applied unless there is a valid reason for this rule being circumvented. Sections using the expression *should* are considered good practice and usually improve the quality of the code. However, software developers are not obliged to conform to this kind of rule but are encouraged to do so. The least restrictive expression used in this document is *can*. These parts generally provide only suggestions or general guidelines which may be applied. Software developers are free to choose equivalent or different rules for the sections marked as *can*.

3.3.1.2. General Requirements

All file names, comments, code and documentation must be written in English or have to be based on the English language. Futher, only letters a-z, the underscore character _ or numerals 0-9 must be used.

For enabling compiler independent code, utilization of common language construct are strongly recommended.

The line length of the code should be limited to 80 characters as much as possible. Longer lines tend to be more difficult to read. For this reason, it is strongly recommended to only use a single statement in each line.

Only *soft tabulators* (i.e. a sequence of single white spaces) must be used instead of *hard tabulators* (i.e. tabulator key). The inferred space of hard tabulators are editor dependent and thus the actual indentation is likely to vary. A sequence of single white spaces are uniformly displayed across editors. It is strongly recommended to use a number of white spaces ranging from 2 to 4 for a single indentation. More white spaces make it easier to find blocks in the source code but increases the overall line length.

3.3.1.3. File Naming Convention

File names are made up of a base name, and an optional period and suffix. The first character of the name should be a letter and all characters (except the period) should be lower-case letters and numbers. The base name should be eight or fewer characters and the suffix should be three or fewer characters (four, if you include the period). These rules apply to both program files and default files used and produced by the program (e.g., ”foobar.sav”).

In addition, it is conventional to use Makefile (not makefile) for the control file for make (for systems that support it) and ”README” for a summary of the contents of the directory or directory tree.

3.3.1.4. Documentation

General: All developed source code has to contain appropriate comments to simplify maintenance and to reduce the required time for other software developers to understand the code. For this reason, meaningful comments have to be written which clearly state the purpose of the following code instead of repeating the code in textual form (e.g. This is an assignment). The following sections cover the parts of the software which require comments. Further comments can be added as seen fit.

Doxygen: Variables and function prototypes should be commented with a doxygen compatible comment syntax to easily create a class documentation for the project. The syntax looks like:

```
/// Brief description.
/** Detailed description. */
```

Note: Doxygen comments are currently not used but are added for forward compatibility.

Standard Top Comment: Each source file must contain a standardized comment at the start of the file, which contains important information. 3.1 shows the structure of an exemplary header comment. The author must provide information for each entry (marked with <>). The *Modified* entry has to be updated each time something has been changed on the current version of the module. Each file has to contain information about the license for this file. Since *ASTERICS* is an open framework, *GNU GPL* is the most common license, however, a different license can be chosen.

```
/**
----------------------------------------------------------------------
--  This file is part of the ASTERICS Framework.
--  (C) <year> Hochschule Augsburg, University of Applied Sciences
----------------------------------------------------------------------
-- File:           <file_name>.c/h
--
```

```
-- Company:         Efficient Embedded Systems Group
--                  University of Applied Sciences, Augsburg, Germany
--                  http://ees.hs-augsburg.de
--
-- Author:          <main_author> [<year>], [<second_author> <year>]
--
-- [Modified:        <modification_author> - <year>: <description>]
--
-- Description:     <Detailed information about the purpose of this
--                  module>
--
----------------------------------------------------------------------
--  <License text>
----------------------------------------------------------------------
--! @file <file_name>.c/h
--! @brief <concise description about the purpose of this module>
----------------------------------------------------------------------
*/
```

Listing 3.1: *ASTERICS* header file

3.3.1.5. Naming Conventions

Pointer Declarations: The pointer qualifier "*" should be written at the variable name, instead of the type. This prevents misreading when multiple variables are declared in one line like:

```
CORRECT:    char *s, *d, *o;  // All variables are pointers
WRONG:      char* s, d, o;    // Only s is a pointer
```

This also affects function parameters.

Typedefs and Structs: Typedefs are an easy way of creating a synonym for data types and change them later if needed. To identify typedefs easily, they have to be named with a *"_t"* suffix. When using a typedef on a struct a suffix *"_s"* has to be appended to the typedef name to differentiate it from typedefs of simple data types. Struct names without typedefs do not require the suffix, as they have to be explicetly referenced using the *struct* keyword, though it is recommended to keep a consistent coding style.

Constants, Enumerations and Macros: Constants have to be added with the *#define* feature of the C preprocessor. Symbolic constants make the code easier to read. Defining the value in one place also makes it easier to administer large programs since the constant value can be changed uniformly by changing only the define. The enumeration data type is a better way to declare variables that take on only a discrete set of values, since additional type checking is often available.

Constants, Enumerations and Macros must be named with capital letters. When the name has more than one word, the words should be separated with underscores to guarantee the readability.

Functions: Words in function names have to be in written in lowercase and seperated with underscores.

3.3.2. VHDL Code

by Alexander Zöllner

3.3.2.1. Purpose

This chapter defines rules and requirements for maintaining and developing VHDL code for the *ASTERICS* framework. The main emphasis hereby lies on layout and naming convention of VHDL source code, files and directories involved. Further, guidelines for VHDL modeling are covered to some extent as seen fit.

The purpose of this document is to enable good comprehensibility of the developed VHDL models across various software developers and increase its general quality. Thus, reducing the initial time required for acquiring understanding of a given VHDL model and simplify its maintenance as well as accelerate integration of newly developed VHDL models into the *ASTERICS* framework.

The given requirements outlined in this document are based on the VHDL-93 standard as common ground.

Since VHDL is the prevalently used language throughout the *ASTERICS* framework, Verilog is not explicitly covered. However, similar rules are recommended.

3.3.2.2. Conventions

This chapter uses the expressions *must/has to*, *strongly recommended*, *should* and *can* to give information about the relevance of the coding convention in question. Rules or sections marked as *must* are binding and are to be applied without exception. *Strongly recommended* parts are to be always applied unless there is a valid reason for this rule being circumvented. Sections using the expression *should* are considered good practice and usually improve the quality of the code. However, software developers are not obliged to conform to this kind of rule but are encouraged to do so. The least restrictive expression used in this document is *can.* These parts generally provide only suggestions or general guidelines which may be applied. Software developers are free to choose equivalent or different rules for the sections marked as *can.*

3.3.2.3. General Requirements

All file names, comments, code and documentation must be written in English or have to be based on the English language. Further, only letters a-z, the underscore character _ or numerals 0-9 must be used.

For enabling compiler independent code, utilization of common language construct are strongly recommended.

The line length of the code should be limited to 80 characters as much as possible. Longer lines tend to be more difficult to read. For this reason, it is strongly recommended to only use a single statement in each line.

Only *soft tabulators* (i.e. a sequence of single white spaces) must be used instead of *hard tabulators* (i.e. tabulator key). The inferred space of hard tabulators are editor dependent and thus the actual indentation is likely to vary. A sequence of single white spaces are uniformly displayed across editors. It is strongly recommended to use a number of white spaces ranging from 2 to 4 for a single indentation. More white spaces make it easier to find blocks in the source code but increases the overall line length.

3.3.2.4. Name Style Rules

The following table 3.1 shows the requirements on identifiers used in the *ASTERICS* framework. Using meaningful non-cryptic identifiers are strongly recommended. Except for *generics*, all identifiers have to be written in lower case, using the underscore character for separation to increase readability. Camel case must not be used.

Description	Extension	Example
`General Signal`	prefix s_	s_load_address
`Signal Inferring Register`	prefix r_	r_memory_addr
`Constants`	prefix c_	c_bit_per_pixel
`Types`	suffix _t	filter_t
`Generics`		DIN_WIDTH
`Keywords`		downto
`Low Active Signal or Variable`	suffix _n	s_reset_n

Table 3.1.: Mandatory extensions used for signals, constants, ...

The following table 3.2 gives a recommendation for the naming convention of ports. If a given port has a counterpart with a different direction, the corresponding identifier has to be used (e.g. data_in, data_out).

Description	Extension	Example
`Input Port`	suffix _in	data_in
`Output Port`	suffix _out	data_out
`In/Out Port`	suffix _inout	sda_inout

Table 3.2.: Recommended extensions used for ports

The software developer is strongly encouraged to use meaningful labels for all sequential blocks, such as process or generate statements. Table 3.3 shows the identifier for the corresponding labels.

Description	Extension	Example
`Process`	prefix p_	p_data_counter
`Generate`	prefix gen_	gen_instantiate_counters
`Loop`	prefix l_	l_sample_input

Table 3.3.: Label prefix for process, generate, ...

The entity name has to use the prefix *as_* if the corresponding module is meant to be used in an *ASTERICS* IP core.

The architecture name has to be *behavior*, *structure*, *rtl* or *testbench* according to the content of the architecture. It is possible for an entity to contain more than one architecture. In this case, the architecture name has to be extended by a meaningful identifier (e.g. rtl_sim, rtl_synth).

All source files of a module which are related to the *ASTERICS* framework have to start with the prefix `as_` if they are meant to be used in an *ASTERICS IP core*. Further, the file name has to contain a short description of the purpose of the module. If the module instantiates submodules, the submodule is strongly recommended to contain the name of the module in its name. The following example 3.2 shows some modules with their corresponding files. In this case the indented names are submodules to the previous module. Since `as_memreader` and `as_memwriter` both use the same submodule, the submodule does not contain *reader/writer* in its name. VHDL files must have the ending *.vhd* or *.vhdl*.

```
as_invert.vhd
as_nitra.vhd
  as_nitra_ctrl_unit.vhd
as_memreader.vhd
  as_mem_address_generator.vhd
as_memwriter.vhd
  as_mem_address_generator.vhd
```

Listing 3.2: *ASTERICS* module file names

A testbench file has to be named after the module which is tested. If more than one module is tested, the name of the top-level module has to be chosen. Testbench files have to be suffixed with *_tb*.

Package files must contain the prefix *as_*, the suffix *_pkg* and the package name has to match the file name containing the package without the file type extension.

3.3.2.5. Documentation

All developed source code has to contain appropriate comments to simplify maintenance and to reduce the required time for other software developers to understand the code. For this reason, meaningful comments have to be written which clearly state the purpose of the following code instead of repeating the code in textual form (e.g. ”This is an assignment”). The following sections cover the parts of the software which require comments. Further comments can be added as seen fit.

Important code sections require a doxygen compatible comment syntax, which consists of *–!*.

Each source file must contain a standardized header, which contains important information about the file. The following code snippet 3.3 shows the structure of an exemplary file header. The author must provide information for each entry (shown as ⟨ ⟩). Each file has to contain information about the license for this file. Since *ASTERICS* is an open framework, *GNU GPLv3* is the most common license, however, a different license can be chosen.

```
------------------------------------------------------------------------
--  This file is part of the ASTERICS Framework.
--  (C) <year> Hochschule Augsburg, University of Applied Sciences
------------------------------------------------------------------------
-- File:            <file_name>.vhd
-- Entity:          <entity_name>
--
-- Company:         Efficient Embedded Systems Group
--                  University of Applied Sciences, Augsburg, Germany
--                  http://ees.hs-augsburg.de
```

```
--
-- Author:          <main_author>, [<second_author>]
--
-- [Modified:       <modification_author> - <date>: <description>]
--
-- Description:     <Detailed information about the purpose of this
--                  module>
--
----------------------------------------------------------------------
--  <License text>
----------------------------------------------------------------------
--! @file <file_name>.vhd
--! @brief <concise description about the purpose of this module>
----------------------------------------------------------------------
```

Listing 3.3: *ASTERICS* header file

The description of the entity has to be partially covered in the file header by describing the purpose of the module. The entity section has to contain comments for each generic and port by describing their purpose. If ports can be grouped to a common interface, it is sufficient to name the corresponding interface. The following code 3.4 shows exemplary comments for the as_invert module.

```
entity as_invert is
    generic (
        -- Width of the input and output data port
        DATA_WIDTH : integer := 8
    );
    port (
        clk          : in  std_logic;
        reset        : in  std_logic;
        ready        : out std_logic;

        -- AsStream in ports
        vsync_in       : in  std_logic;
        vcomplete_in   : in  std_logic;
        hsync_in       : in  std_logic;
        hcomplete_in   : in  std_logic;
        strobe_in      : in  std_logic;
        data_in        : in  std_logic_vector(DATA_WIDTH - 1 downto 0);
        data_error_in  : in  std_logic;
        sync_error_in  : in  std_logic;
        stall_out      : out std_logic;

        -- AsStream out ports
        vsync_out       : out std_logic;
        vcomplete_out   : out std_logic;
        hsync_out       : out std_logic;
        hcomplete_out   : out std_logic;
        strobe_out      : out std_logic;
        data_out        : out std_logic_vector(DATA_WIDTH - 1 downto 0);
        data_error_out  : out std_logic;
        sync_error_out  : out std_logic;
        stall_in        : in  std_logic;

        --! Slave register interface:
        --! Control registers. SW -> HW data transport
        slv_ctrl_reg    : in slv_reg_data(0 to 0);
        --! Status registers. HW -> SW data transport
        slv_status_reg : out slv_reg_data(0 to 0);
        --! Aquivalent to a write enable signal
        slv_reg_modify : out std_logic_vector(0 to 0);
        --! Slave register configuration table.
        slv_reg_config : out slv_reg_config_table(0 to 0)
    );
end as_invert;
```

Listing 3.4: *ASTERICS* entity comments

Blocks, such as *generate* statements, or processes must be preceded by a comment section, which gives detailed information about the purpose of the following code. The following code 3.5 shows comments for a process which detects a synchronization error inside the module. The first three lines are optional, however, grouping logical units and precede them with a distinctive comment makes it easier to find them.

```
---------------------------------------------------------------------
-- Process for Detecting Synchronization Error
---------------------------------------------------------------------
    --! A sync_error is detected when a strobe is received despite
    --  the fifo buffer
    --! being full. The error can only be lifted by resetting the
    --  memwriter.
    p_sync_error : process(clk)
    begin
        if rising_edge(clk) then
            if reset = '1' or s_reset_soft = '1' then
                s_sync_error <= '0';
            else
                if strobe_in = '1' and s_fifo_full = '1' then
                    s_sync_error <= '1';
                end if;
            end if;
        end if;
    end process;
```

Listing 3.5: Exemplary comment for process

State machines are a special type of data processing and usually comprises one or more processes. For this reason, each state of the state machine has to be commented individually by describing its purpose and listing the condition(s) to be met to transition to the following state. The following code 3.7 shows the comment section for the `s_idle` state of the `as_memwriter` state machine.

```
        --! Default state of as_memwriter if hw module is ready
        --! for receiving order to conduct writing memory data.
        --! During idle state config parameters are continuously
        --! copied into shadow registers to begin memory access
        --! right away after receiving "go" signal. The state
        --! machine transitions to "s_init" after receiving a
        --! "go" signal.
            when s_idle =>
                s_mem_sm_done <= '1';
                s_mem_sm_save_config_reg <= '1';
                if s_mem_sm_go = '1' then
                    mem_sm_next_state <= s_init;
                    end if;
```

Listing 3.6: Exemplary comment of a state

Adding comments for declaration of signals, variables and constants is strongly recommended to make their purpose transparent. Further, related declarations should be grouped together to make it easier to find them. The following code 3.7 shows signal declarations of the `as_iic` module.

```
  --! Used to locally store the "scl_div" signal and compare with
  --  "r_scl_counter"
  signal r_scl_div_local : std_logic_vector(SCL_DIV_REGISTER_WIDTH-1
                                            downto 0);

  --! A counter running with the system clock, as long as
  --  "enable_sclcntr" is high
  signal r_scl_counter : std_logic_vector(SCL_DIV_REGISTER_WIDTH-1
                                          downto 0);
```

Listing 3.7: Exemplary comment for a declaration

Assignments are strongly recommended to be accompanied by appropriate comments to indicate the purpose of the statement. A tolerated exception is so-called *glue logic* between modules, which is used to map in and out ports between modules. The comments for glue logic may be omitted. The following code 3.8 shows exemplary comments for signal assignments.

```
----------------------------------------------------------------------
-- Status Information for Hardware
----------------------------------------------------------------------
-- Signal previous hardware module being ready for operation
    ready            <= not s_mem_sm_busy;
-- Signal other hardware modules that a sync error has been detected
    sync_error_out   <= s_sync_error;
```

Listing 3.8: Exemplary comment for an assignment

3.3.2.6. Generator specific conventions

The *ASTERICS* chain generator *as-automatics* is used to analyse part of the VHDL files of *ASTERICS* modules. To make sure that as-automatics can precisely identify the modules ports and generics, certain naming conventions should be followed:

- Port and generic name fragments should be delimited using underscores '_'
- When implementing an interface, such as `as_stream`, the interface port names must be used
- When multiples of the same interface is implemented, the port names must be differentiated using suffixes or prefixes, delimited using underscores
- When implementing custom ports, names matching interfaces used in *ASTERICS* should be avoided to mitigate false positives from as-automatics. Alternatively, different delimiters may be used, keeping the generator from fully parsing the port name, effectively avoiding false positives.

Port name examples:

- `activity`: Unproblematic name for a custom port
- `activity_strobe_out`: Possible false positive, `as_stream` uses a port named `strobe`! The name fragments are correctly delimited using underscores, which allows the generator to parse and compare the name.
- `activity-strobe-out`: Less problematic, the generator will not be able to extract the name fragment `strobe`, which might cause a false positive.

3.4. Python Code

by Philip Manke

This section briefly outlines the coding style and rules for Python code within the *ASTERICS* Framework. All code *must* be compatible with Python version 3.5. Python 2 must not be be used.

3.4.1. Coding Style Guidelines

All Python code within *ASTERICS* must follow conventions described in PEP 8 released on Python.org: https://www.python.org/dev/peps/pep-0008/. Where necessary or to increase the readability of short sections of code, rules described in PEP 8 may be purposefully broken. Before checking code into the Git repositories, developers should run the code formatter `autopep8`, which can be found in `pip`.

3.4.2. File Structure

Each Python file must include a standardized header, a template of which is available here, in listing 3.9. Everything marked with pointy brackets (`< >`) must be replaced by values for the respective file. Parts marked in square brackets are optional and may be ommited. This header is somewhat redundand to provide information using both Python's built-in docstrings and Doxygen.

```
# ------------------------------------------------------------------------------
# This file is part of the ASTERICS Framework.
# (C) <year> Hochschule Augsburg, University of Applied Sciences
# ------------------------------------------------------------------------------
"""
<filename>.py

Company:
Efficient Embedded Systems Group
University of Applied Sciences, Augsburg, Germany
http://ees.hs-augsburg.de

Author:
<author> [<year>][, <second author> <year>]

[Modified:]
```

```
[<author> - <year>: <modification>]

Description:
<description of the module/class purpose and functionality>
"""
# -------------------- LICENSE ---------------------------------------------
#
# <license text here>
#
# -------------------- DOXYGEN ---------------------------------------------
##
# @file <filename>.py
# @author <main author[s]>
# @brief <concise description>
# --------------------------------------------------------------------------
```

Listing 3.9: Template for the standard header of a python source file.

In general each Python file should only include a large class, multiple smaller classes, only class-independent functions or constant values. Typically, a Python file should contain less than 1000 lines of code. Developers should use typing hints, where applicable (`example_function(number : int, text : str) -> bool:`).

Part II.

Reference Guide

4. The *ASTERICS* Software Stack

4.1. Overview on the *ASTERICS* Software Stack

by Alexander Zöllner, Philip Manke, Gundolf Kiefer

ASTERICS offers a sophisticated software stack, which allows to conveniently interface any kind of *ASTERICS* -based hardware image processing chain. The software stack is divided into layers, which gradually abstract from the actual hardware. Figure 4.1 shows the layers of the *ASTERICS* software stack, ranging from implementation specifics at the bottom, up to user application software at the top. *Layer 0* comprises vendor and platform specific libraries of the utilized FPGA and CPU, which are included depending on the chosen settings in the *ASTERICS* configuration. The *ASTERICS* configuration contains supported features available to *ASTERICS* , such as the type of operating system.

The *ASTERICS Support Library* (*ASL*), located in *Layer 1*, adopts the settings of the *ASTERICS* configuration, includes required vendor and platform libraries. The *ASL* performs actual accesses to the underlying hardware and thus is the only part of the *ASTERICS* software stack with external dependencies. Unified interfaces are offered to the higher layers, which are unaffected by the settings of the *ASTERICS* configuration. Merely their underlying implementation in the *ASL* are altered.

Layer 2 contains the *ASTERICS* module drivers, which offer methods for accessing the corresponding hardware module and obtaining its status information by using the interfaces provided by the *ASL*. The module drivers also define descriptive aliases for address calculation and proper bit masking.

The `asterics.h` library in *Layer 3* is the resulting library for the application software and includes the *ASTERICS* module drivers as well as the *ASL*. It also contains specific information about the hardware image processing chain implemented on the FPGA, such as absolute addresses of the hardware modules.

The application software in *Layer 4* is able to access the interfaces of each layer directly by including `asterics.h`.

4.2. The *ASTERICS* Support Library (ASL)

by Alexander Zöllner, Gundolf Kiefer

The ASTERICS *Support Library* is the main interface between hardware and application software within the *ASTERICS* framework. It contains all vendor and platform dependencies for performing the actual hardware accesses. Towards software, the *ASL* offers static register-based interfaces to the *ASTERICS* hardware modules of the hardware image processing chain and a number of utilities, such as for synchronization or memory allocation. The *ASL* uses the appropriate implementation for the corresponding functionality, depending on the environment parameter settings within `as_config.h`.

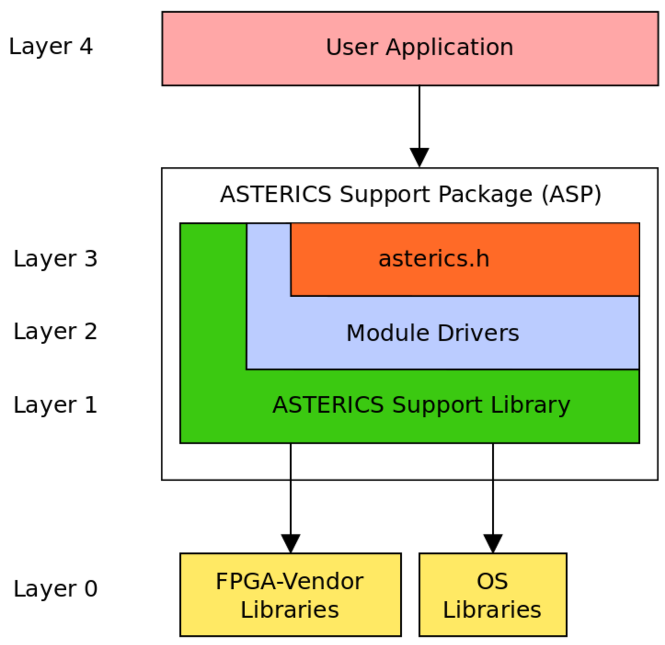

Figure 4.1.: Overview of the *ASTERICS* software stack

Therefore, the *ASL* has to be included across the *ASTERICS* software stack, whenever a piece of software indents to access the hardware or utilizing mechanics which are platform dependent.

4.3. Contents of an *ASTERICS* Support Package (ASP)

by Alexander Zöllner, Philip Manke, Gundolf Kiefer

Table 4.1 shows the contents of an ASTERICS *Support Package*, used for describing a specific hardware image processing chain and choosing the target environment of the *ASP*. If a given hardware image processing chain is to be utilized for several environments, a corresponding *ASP* has to be provided for each one. Although most contents of the *ASTERICS* software stack can be copied for multiple *ASPs* and processing chains, some parts have to be replaced, since they depend on the actual hardware implementation and the environment. The files `asterics.h` and `as_config.mk` are the only ones which are affected. The former has to be updated each time the underlying hardware image processing chain is altered in regards of its modules. This mainly includes adding or removing hardware modules, changing their pre-synthesis parameters or their absolute start address. Therefore, it is recommended to update the *asterics.h* header file each time one of the parameters of the hardware image processing chain is touched. On the other hand, the `as_config.mk` file reflects the settings of the environment, on which the *ASP* and hardware image processing chain is deployed. Prominent parameters are the whether an operating system is utilized or the SoC vendor, who usually provides low level software drivers for interfacing the hardware. Thus, it only needs to be replaced when the target environment changes (which usually occurs less frequent than changing the hardware implementation during development). The remaining files of the *ASP* are left untouched across multiple hardware image processing chains and environments.

File Name	Includes	Description
asterics.h	as_support.h as_<*module*>.h	Library to be included by the user application software. Inlcudes all module driver headers and defines hardware addresses.
as_support.h	as_config.h as_kernel_linux_if.h(*)	The ASTERICS *Support Library* for vendor abstraction and interfacing the hardware. (*) Only included when compiled for Linux kernel or POSIX compliant operating systems.
as_support.c	as_support.h	Contains the function definitions for the ASTERICS *Support Library*.
as_<*module*>.h	as_support.h	Provides macros and interface functions of the associated hardware module. *module* is replaced by the actual name of the hardware module.
as_<module>.c	as_<*module*>.h	Implements the functionality of the module driver.
as_config.h		Contains macros for defining the environment the *ASP* is built for. It is generated from as_config.mk.
as_config.c		Contains an identification number and build date of config.h. It is generated from as_config.mk or by *Automatics*.
as_config.mk		Sets the platform and operating system configuration of the *ASP* Makefile fragment.

Table 4.1.: Contents of an ASTERICS *Support Package (ASP)* for a specific ASTERICS *chain*.

4.4. Transferring Data between Hardware and Software

by Alexander Zöllner, Gundolf Kiefer

4.4.1. Brief Description

The main purpose of the *ASTERICS* framework is enhancing image processing tasks using codesigns of hardware and software. This requires transferring data between both subsystems in a convenient and efficient manner. For accomplishing this task, the memory modules of *ASTERICS* are used for performing the actual data transfer. Since setting up data transfers has to be managed by the software stack, respective methods are provided. These methods vary in their complexity and required overhead expected from the user. The appropriate method can be chosen depending on the requirements of the application.

4.4.2. Manual Data Transfer Management

This method is the most direct way for transferring data by interfacing the corresponding memory module, using their module driver (see Chapter 7.2). Here, a sufficient physically concurrent memory area has to be provided to the corresponding memory module, which is referred to as *buffer* for (intermediately) storing data. The user is supposed to manage the contents of the *buffer* manually, to prevent its under- or overflows as well as inadvertently overwriting data. Further, the status of the memory module has to be read from its hardware registers in order to determine whether a data transfer has been finished (*state/control* register) or which parts of the *buffer* have already been processed (*current hw addr* register). The double buffering scheme of the memory modules for queuing data transfers may be used for utilizing more than one *buffer*.

Managing the data transfers directly gives the user exclusive control, allowing to also implement customized data transfer strategies. However, preventing data loss has to be taken care of explicitly.

4.4.3. POSIX-like Data Transfer Management

For transferring data in a more convenient manner, the *ASTERICS* framework offers POSIX-like interfaces to the user, which are part of the software module `as_memio` (memory input/output). This module includes implementations for *open*, *read*, *write* and *close*, respectively. Instead of having the user manage the data transfers and organizing the *buffer(s)* for the memory module manually, these tasks are carried out by *as_memio*. The user has to simply call *open*, where the actual memory module is referenced. The `as_memio` module handles the internal specifics required for setting up data transfers. These can be requested by calling either *write* for transferring data to the hardware, using an `as_memreader` module, or *read* for obtaining data from the hardware processing chain, using an `as_memwriter`. Here, an opaque structure is provided to `as_memio` (which is obtained by *open*), along with the desired amount of data and a *user buffer*. This *user buffer* is used for providing the data to be transferred or receiving data from `as_memio`. Once the request from the user has been served, the actual number of transferred bytes are returned to the user, which may be equal or less than the requested number. Further, the *user* buffer contains the amount of data, which has been served by the `as_memio` module for *read* calls. For *write* calls, the *user buffer* is left untouched and may be used immediately by the user again after the function returns. The `as_memio` module internally manages the data for either direction and interfaces the associated memory module accordingly for transferring the data. Provided the hardware processing chain supports the `STALL` mechanic (see Chapter 5.3), the `as_memio` module can guarantee that no data is lost.

4.5. The Linux Kernel Driver

by Alexander Zoellner

4.5.1. Brief Description

Within *ASTERICS* great emphasis is put on its usability for developing new image processing applications in a fast and convenient manner. Utilizing an operating system is a common practice since it already provides a great deal of functionality, such as a network stack and memory management. Further, required software is either already available or can be easily installed by using the package manager. Being able to seamlessly integrate *ASTERICS* into own applications

As Linux is commonly used for embedded applications, the Linux character device driver `as_driver` has been developed for *ASTERICS* . This driver is able to operate with any *ASTERICS* -based image processing chain by providing a set of interfaces between hardware and software. *ASTERICS* -chains can be exchanged on the FPGA without having to reload or recompile `as_driver`. The driver covers standard POSIX file operations, mapping memory regions to user as well as basic register-based hardware accesses. The driver provides methods for altering the interfaces to hardware at runtime to cater to any *ASTERICS* -chain.

4.5.2. Architecture

Figure 4.2 shows the principle architecture of `as_driver`. It consists of the main parts *device class*, *device array*, *file operation structures*, *Init* and *Exit*. The latter two are methods which are the constructor and destructor of the kernel module, respectively. They are called when the kernel module is loaded to the kernel or unloaded. *Init* performs the minimal required amount of initialization of the kernel module, such as registering it to the kernel and publishing its file operations. For this reason, a *device class* is created, which consists of a *major number*, which refers to the kernel module itself and a number of *minor numbers*. Each *minor number* is associated with a specific device of the kernel module, whereas the *major number* indicates the responsible device driver for the device. As shown in the figure, a number of *minor numbers* are requested but not published immediately, indicated by *empty* slots in the *device class*. The *ASTERICS* device driver organizes its devices in a *device array* of a static size. Devices may be added or removed from the *device array*. Slots which are not yet occupied by a device are indicated by *uninitialized*. Each device is associated with a *minor number* and a *file operation structure*. The five *file operation structures* are the actual device types available to `as_driver`. Each structure defines a set of *methods*, which are used to overwrite the default file operation methods provided by the kernel. This is accomplished by linking to one of these *file operation structures* when initializing the device. As shown by the `as_memio` device, devices of the same type also point to the same *file operations structure* and use its *methods*. The first device, namely the `as_control` device, is created by *Init* and the first *minor number* is used. Additional devices can be added or removed by using the *methods* of the `as_control` device, which publishes the device to the kernel by adding the associated *minor number* to the *device class* and linking to the appropriate *file operation structure*. *Exit* on the other hand, deletes all allocated parts of the *ASTERICS* device driver and unregisters the kernel module and all its devices. Resources obtained by the kernel module during its lifetime are released to the operating system again.

Up to this point, devices contained in the *device array* are only known by the kernel but not yet accessible by the user application. This requires to link the devices to the file system of the operating system in order to perform operations on the it. A device can be published to the user space by creating a *device node*, which is an entry point on the file system. A *device node* is associated with a certain device, by specifying its *major* and *minor number* upon creating the node. As aforementioned, the *major number* is used to tell the kernel which kernel module it should use when calling the file operations of the device. The *minor number* is used by the kernel module to identify the actual device to be accessed. For creating the *device node*, a path and a name on the file system has to be chosen where the node should appear. When working with Linux, the location */dev/* is usually used for devices. Once the *device node* has been created, it can be used for the path argument for the open file operation to access the device and subsequently perform additional actions by calling the appropriate file operations. Since the *device node* is only a link to a device, it can also be created before the actual device exists, provided that the *major* and *minor number*, which is going to be used, is already known at this point. Trying to access the *device node*, without the actual device existing, will fail for obvious reasons. The node can be used as soon as the associated device has been published to the kernel. Naturally, if the device is removed, the node will cease to perform any accesses to the device.

The structure of `as_driver` allows to create additional devices at runtime and therefore can be used for any *ASTERICS* -based image processing chain implemented on hardware. Moreover, devices can also be deleted at runtime, with the kernel module displaying similar behavior as if it has just been loaded to the kernel. Therefore, the processing chain can also be replaced at runtime, by simply instructing the kernel module to delete all devices and subsequently creating new ones.

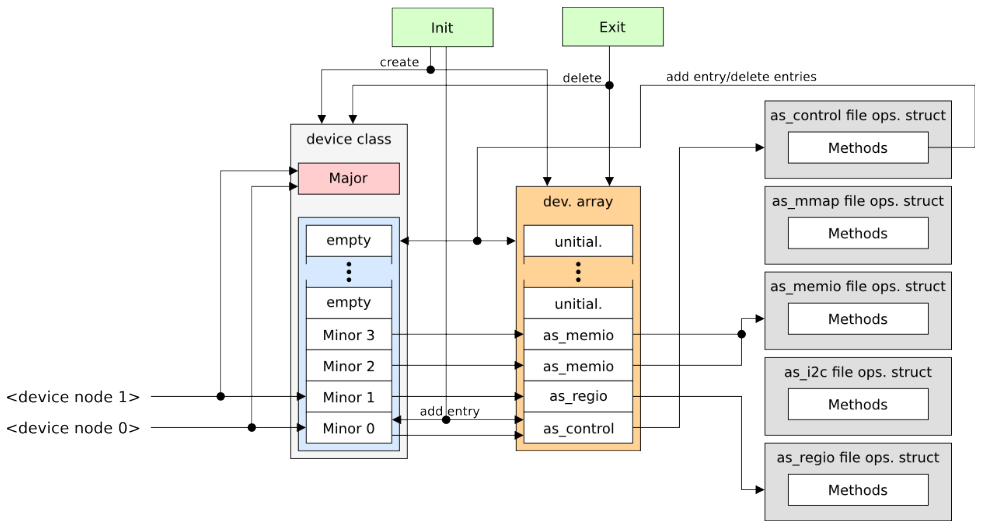

Figure 4.2.: Architecture of the *ASTERICS* Linux kernel driver (`as_driver`)

Table 4.2 lists the files of `as_driver` and their meaning. They can be found at "tools/as-linux/src/kernel_module/asterics-driver".

Name	Type	Description
`as_driver`	c source file	Implementation of the *ASTERICS* device driver.
`as_driver`	c header file	Compile time parameters and data structure of the *ASTERICS* device driver.
`as_linux_kernel_if`	c header file	Data structures and commands for `ioctl/unlocked_ioctl` and device types used by the `as_control` device.
`as_config`	c header file	Flags used for determining the features and environment the software has been compiled for. It is used for determining the appropriate implementation for functions provided by the *ASTERICS* software stack.
`Makefile`	GNU makefile	Builds the kernel module.

Table 4.2.: Files of the *ASTERICS* device driver

4.5.3. Compile-Time Options

The `as_driver` offers a set of configuration parameters, which can be set at compile-time and therefore take effect, once the device driver is loaded. Table 4.3 lists the available parameters. The device driver manages its devices in a list, with a fixed number of entries. The parameter `NUM_MAX_DEVICES` is used to define the number of entries in the list, which correlates to the maximum number of devices which can be created by the `as_control` device. The number of devices has to be at least set to two, since one entry will be already occupied by the `as_control` device. As for the maximum number of entries, it is best to make use of the define found in the library *kdev_t.h*, which defines the maximum available amount of supported minor numbers for device drivers. Each device requires a minor number in order for the kernel to distinguish between them.

The parameter `TIMER_INTERVAL` is used to define the interval in jiffies for generating a timer interrupt. The actual interval depends on the settings for *HZ* of the platforms. For embedded platforms, it usually defaults to 100 Hz, which means the interval has a granularity of 10 ms. Although the kernel provides an interface for converting a desired time in milliseconds to the appropriate number of jiffies, it cannot forgo the configured granularity of the interval. This means, the resulting time may not match the actual requested time by the user, which could lead to confusion. Thus, the parameter `TIMER_INTERVAL` uses jiffies to clearly indicate its platform dependency. An appropriate comment has also been added to the header file of the device driver to state this circumstance. Depending on the expected amount of data to be transferred between software and hardware, this value can be increased or lowered accordingly. For a low amount of data, the user may consider to increase the timer value, since calls to read or write of the `as_memio` device may block for an extended period of time, until data becomes available again. Since the interrupt is also used to wake up any sleeping processes, they might block immediately again, since no data has been become available between two consecutive interrupts.

Name	Type	Range	Description
`NUM_MAX_DEVICES`	unsigned	2 - MINORMASK	Maximum number of devices available to the driver
`TIMER_INTERVAL`	unsigned	1 - 4294967295	Timer interval in jiffies

Table 4.3.: Configuration options for the ASTERICS device driver

4.5.4. Register-based IO Device

In order to access the physical addresses of the hardware registers, they have to be mapped to a virtual kernel address area first. Listing 4.3 shows the functions, which have to be called for being able to access the physical address region. The first function, `request_mem_region` is used to reserve a named address region from the kernel. By calling this function the driver tells the kernel, that it is going to use this address region. No actual mapping is performed at this point, only a reservation request. This prevents other drivers from mapping the same address region and thus competing accesses. The first parameter `start` specifies the physical address, where the requested region starts, with a size of `n` bytes. The last parameter `name` provides a pointer to a string, containing the name of the region. If the request has failed, a NULL value is returned by these function, otherwise a non-NULL value.

The function `ioremap` is used to perform the actual mapping of the memory region. It also requires the physical start address `start` of the region as well as its `size` in bytes. The return value is a virtual kernel address, which can be used to perform the actual accesses to hardware.

The mapping of the memory region is performed by the `as_control` device upon creating the `as_regio` device. Similarly, unmapping and releasing the memory region is either performed by the `as_control` device or at the point `as_driver` is unloaded.

```
struct resource * request_mem_region (unsigned long start, unsigned long n, const

void *  ioremap (unsigned long phys_addr, unsigned long size)
```

Listing 4.1: Functions for allowing to access physical addresses

The following Listing 4.2 shows the functions used by the `as_regio` device of the *ASTERICS* device driver for accessing the hardware registers of the hardware. For obtaining the currently stored data in the hardware register, `ioread32` is used. As the name suggests, it is used to read the value from the register at `addr`, which is the virtual kernel address, previously mapped by calling the aforementioned functions. The return value is the current register content. For writing data to the register, `iowrite` is used, which takes two parameters, the value to be written (`val`) and the virtual kernel address `addr`. It is worth to note, that the currently used platforms, on which the device driver is used, are exclusively 32 bit systems and the hardware registers have also a size of 32 bits.

```c
unsigned int ioread32(void __iomem *addr)

void iowrite32(u32 val, void __iomem *addr)
```

Listing 4.2: Functions for accessing hardware registers

The `as_regio` device provides only an implementation for the *unlocked_ioctl* method, whereas for open and close the default implementations provided by the kernel are used. Similar to the `as_control` device (Chapter TBD), the `cmd` parameter is used to inform the device, whether the method has been called by user or kernel space, in order to copy the data of `arg` appropriately. The `unlocked_ioctl` method can also be called by kernel space, since other devices of the *ASTERICS* device driver are also required to configure the hardware. Instead of performing accesses to the hardware registers on their own, they make use of the `as_regio` device. The `arg` parameter of `unlocked_ioctl` is a pointer to a `as_ioctl_params_t` structure, which is part of `as_linux_kernel_if.h`. The `cmd` parameter of this structure can either be `AS_IOCTL_CMD_READ` or `AS_IOCTL_CMD_WRITE`. In order to access the register correctly, the field `address` has to hold the physical address of the register. This address may be obtained by the tools used for implementing the hardware design (e.g. Vivado). For write accesses, the parameter `value` is written to the register, whereas for read accesses, the value of the register is directly returned by *unlocked_ioctl*, instead of copying it to the field of the structure. The `user_addr_start` is not used by the `as_regio` device and the user may decide to not explicitly assigning a value to it.

The user calls the associated *ioctl* method with the same parameters.

4.5.5. I2C Device

The *i2c device* works identically to the `as_regio` device and uses the same kernel mechanisms for obtaining a memory region and accessing the hardware registers. A separate device has been added to the *ASTERICS* device driver, to be able to map the hardware addresses of the used hardware registers to a different address region, which is not adjacent to the one used for the `as_regio` device. The actual functionality for the I2C uses the `as_i2c` module driver, which utilizes the device driver for performing accesses to its hardware registers.

4.5.6. Memory IO Device

The `as_memio` device utilizes the `as_memio` module driver for conveniently transferring data between application software and FPGA. For being able to transfer data, the `as_memio` device has to be associated with a memory module, which in turn is forwarded to the `as_memio` module driver. In order to use a specific `as_memio` device right away, this task is carried out upon creating the device with the `as_control` device. Here, the *base address*, *memory bus interface width* and *direction* has to be provided. The *base address* is the address of the first hardware register used by the corresponding memory module, which is required for configuring the module correctly. This address can usually be obtained by the tool used for implementing the processing chain on the FPGA (e.g. Vivado by Xilinx).

Similarly, the *interface width* is needed for aligning the data correctly for its transfers. The memory modules are synthesized for a certain bit width, which is also used for its port towards other hardware modules. For this reason, *byte enables* are currently not supported, which results in having to transfer a multiple of the *interface width* of bytes.

Since the *direction* of the data flow is determined by either using an `as_memreader` (to FPGA) or a an `as_memwriter` (from FPGA) module, it has also to be provided to the `as_memio` device. Although the `as_memio` module driver supports both directions, only one can be used at a time, since an instance of the driver only manages one memory module at a time.

The file operation structure for the `as_memio` device provides implementations for five methods, namely `open`, `read`, `write`, `unlocked_ioctl` and `close`. Principally, the `open` method performs a number of checks and sets up the `as_memio` module driver for the specific memory module.

In a first step, the *device array* is iterated to find the `device_data_t` structure (part of `as_driver.h`), which represents the requested `as_memio` device. Multiple instances of the `as_memio` module driver using the same memory module conflict with each other, due to relying on status information of the module regarding the current data transfer. Therefore, only one instance of a given `as_memio` device is allowed. This is guaranteed by using the variable `busy` of the `device_data_t` structure, which is of type *atomic_t*. Since the access to the variable is atomic, only one process can successfully acquire the `as_memio` device.

After having acquired the corresponding `as_memio` device, the directions `flags` provided upon `open` are compared to the one set at the point the device has been created. In this way, the user is informed if the wrong device has been accidentally requested, e.g. data is to be transferred from memory to the FPGA but the device only supports inverse direction. If this has been the case, the `as_memio` device is released again and provides an error message to the user, pointing out this mismatch.

Otherwise, `as_memio_open` of the `as_memio` module driver is called with a set of configuration parameters. Except for the *interface width*, the default settings defined in the header file of the `as_memio` module driver are used. The allocation of the *Ring Buffer* and the *Buffer Handler* is carried out by the module driver internally, without requiring the device driver to explicitly allocate structures on its own. The module driver returns a pointer to the `struct as_memio_file_s` structure, which is used by the module driver for managing the memory module. The pointer is assigned to the `memio_file` field of the `device_data_t` structure of the device.

As the `as_memio` device supports blocking and nonblocking data transfers, the presence of the `O_NONBLOCK` flag is checked. If this flag has not been provided upon calling `open`, a *wait queue* is set up for the device, allowing it to sleep if the request number of bytes cannot be served right away. Lastly, the variable `memio_active` is set to inform the interrupt logic to serve this device. Figure 4.3 summarizes the steps performed by the *open* file operation method.

For actually transferring data between main memory and the FPGA, the methods `read` and `write` are used. In order to prevent race conditions of multiple processes trying to read or write data at a given time, using the same `as_memio` device, a *mutex* is used. For this purpose, the `device_data_t` structure of the device provides the variable `access_lock`. Similar to `open`, the provided direction flag is evaluated, since only either of the two file operations is supported, depending whether a `as_memreader` or `as_memwriter` module

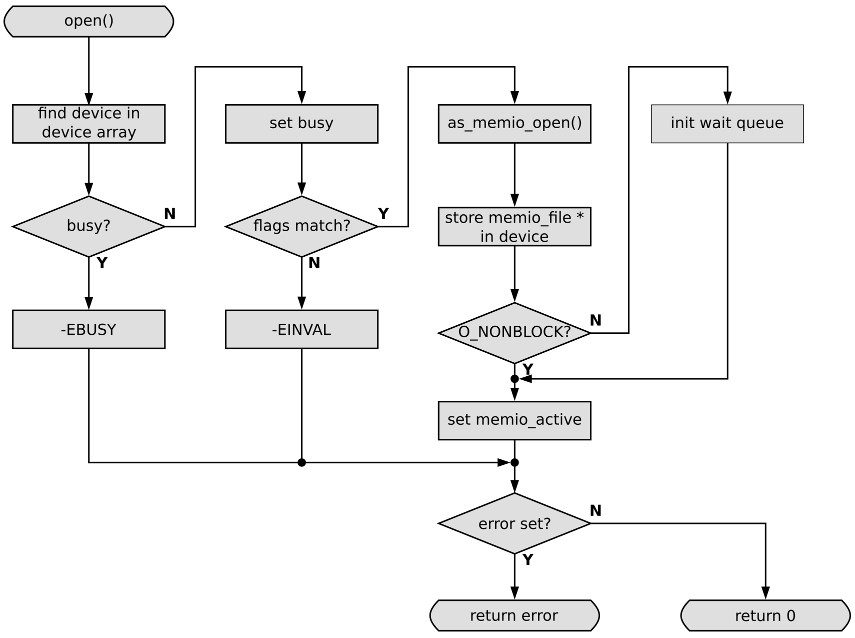

Figure 4.3.: Processing steps performed by the `open` method of the `as_memio` device.

is associated. If the requested file operation is not supported by the specific `as_memio` device, an error message points out the mismatch and a negative value is returned. For this reason, the user has to evaluate the return value of the used function, at least for the first call.

Subsequently, the presence of `O_NONBLOCK` is checked. If the flag has been provided, the specified number of bytes and the *User Buffer* is passed to the `as_memio_[read/write]` function of the `as_memio` module driver. The *Buffer Handler* tries to serve the request as far as possible by transferring data between the *User Buffer* and the *Ring Buffer* and configuring the associated memory module accordingly. Since the data cannot be directly copied between user and kernel space, the function *copy_[to/from]_user* has to be used. Therefore, the module driver checks whether it has been compiled for the Linux kernel or for a bare-metal application, using the setting provided by `as_config.h`. The actually transferred number of bytes is returned to the `as_memio` device, which is equal or less than the requested number of bytes. This number is then passed to the user, returning immediately without blocking even if the requested number has not been met. The POSIX standard states, that a negative return value shall be passed to the user, in case no data has been transferred at all and the device would block with the `O_NONBLOCK` flag being set. However, since the `as_memio` device does not block under any circumstance if the `O_NONBLOCK` flag has been provided, no error code is returned in this case. The `read/write` functions simply returns with a "0" for the number of bytes, which have been transferred. The caller of the corresponding function is expected to evaluate the return value and call the function again, if the requested amount has not been served entirely.

If the `O_NONBLOCK` flag is absent, the `as_memio` device performs the same call to the `as_memio` module driver. If the module driver has been able to perform the data transfer completely, the method behaves in the same manner as if the flag had been provided. However, in case the transferred number of bytes is less than specified, the `as_memio` device sets up the condition variable `wake_up_cond` and sets the flag `register_intr` before blocking by calling the function `wait_event_interruptible`. The processor stops executing the blocking process. When the interrupt handler wakes up the process again by setting the condition variable appropriately and calling the function `wake_up_interruptible`, the device driver calls `as_memio_[read/write]` function again with the remaining bytes to be transferred. If the request has been served entirely, the file operation method of the `as_memio` device exits. Otherwise, the above mentioned procedure is repeated. When exiting, the mutex is released again. Figure 4.4 summarizes the processing steps performed by the `read` and `write` method of the `as_memio` device.

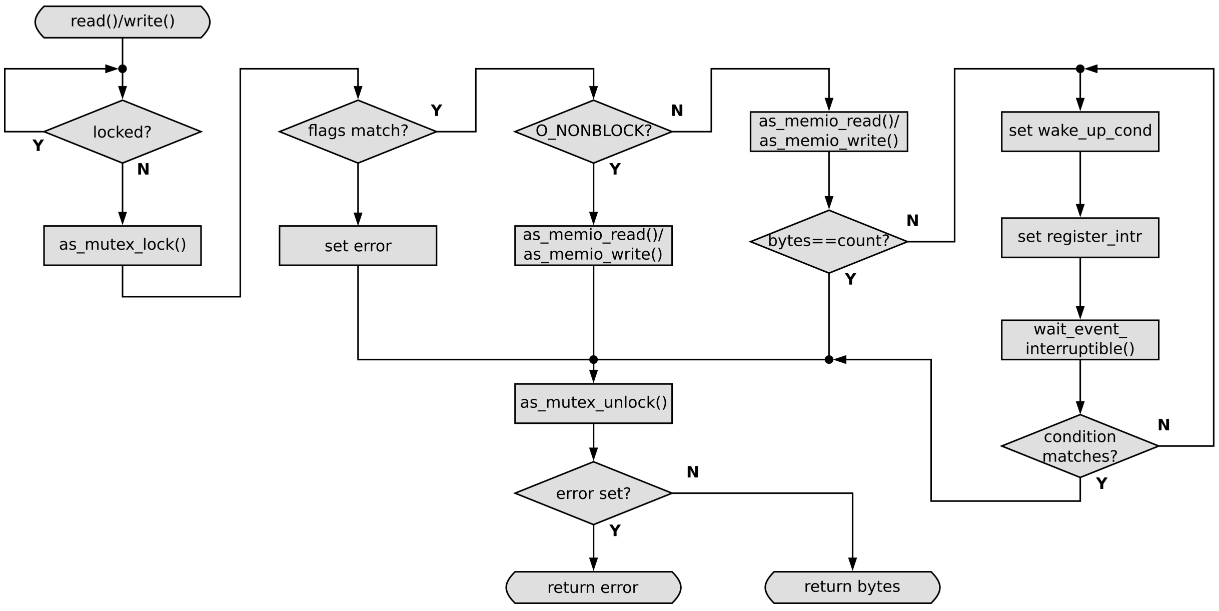

Figure 4.4.: Processing steps performed by the *read* and *write* method of the `as_memio` device.

The `unlocked_control` method of the `as_memio` device is used to trigger the *Buffer Handler* of the `as_memio` module driver to check whether there is data within its *Ring Buffer* to be transferred. The `as_memio_hw_update` function of the module driver is used, by providing the pointer to the `memio_file`. Although this function is called by the module driver for every read and write request, it may occur that two calls to this function are required for transferring all data. This is caused by the discrepancy of the differing access types of the *Buffer Handler* and memory module to the *Ring Buffer*. The former writes or reads circularly to or from the *Ring Buffer*, i.e. if the upper boundary is reached, it continuous at the lower boundary automatically. The memory module, however, can only perform data transfers on physically concurrent memory addresses. If a wrap around is required, the *section* of the memory module has to be configured up to the upper boundary of the *Ring Buffer* and the second one starting at the lower boundary again. Usually, an explicit call to the `unlocked_ioctl` method is not required, since the interrupt handler of the *ASTERICS* device driver regularly calls `as_memio_hw_update` for its `as_memio` devices, which are currently being used (see Chapter 4.5.9).

The `close` method of the `as_memio` device resets the two variables, which are used for the interrupt logic of the device driver, `memio_active` and `register_intr`. In order to return the acquired resources by the `as_memio` module driver back to the kernel, it calls the function `as_memio_close`. Additionally, the module driver performs a reset on the associated memory module. This terminates all ongoing data transfers, since the allocated *Ring Buffer* is deleted and thus the memory area is no longer valid. The `as_memwriter` assumes a passive behavior, which prevents it from blocking any associated hardware processing chain.

4.5.7. Memory Mapped IO Device

The `as_mmap` device circumvents having to copy data on memory by utilizing a dedicated memory area, which is published to the user. This memory can be shared between hardware and software. Since the Linux operating system utilizes virtual address spaces, actual physical memory cannot be accessed from user space in the same way as its virtual addresses. However, this limitation can be lifted to some degree, by utilizing mechanisms provided by the kernel to publish certain areas of physical memory to the user. This is accomplished by implementing the file operation method `mmap` in a device driver for mapping a physically concurrent area of memory into the virtual address space of the user. For this reason, the `as_mmap` device has to provide a memory area which meets the requirements for being able to be mapped. Regarding the lifetime of a device driver, the required memory can be acquired at several stages, such as at the time being loaded as kernel module, upon device creation or when the corresponding device is actually used. Since `as_driver` aims at being operable for any kind of image processing chain without having to reload the kernel module, the actual number of devices required by the user cannot be determined at the time it is being loaded to the kernel. As the aforementioned memory areas are only utilized by devices of the `as_mmap` type, a single memory area is allocated for each `as_mmap` device upon creation, using the `as_control` device. The size of the memory area is configurable by providing the appropriate parameter to the `as_control` device. The memory area is acquired by using the kernel function `__get_free_pages`, which allocates a physically concurrent amount of memory and returns the start address of it, i.e. a virtual kernel address. This address is stored within the device to be used later on for the file operation methods. As a side note, only sizes up to 4 MB have been used. Allocations are to the power of two. For preventing memory leaks, the memory has to be released when the corresponding device is no longer needed, i.e. when it is deleted.

As already stated, it would also be possible to allocate the memory when using a file operation method, such as `open`. However, allocating memory can take quite some time, especially when trying to acquire larger areas of memory. Additionally, it tends to get worse the longer the system runs, as memory gets fragmented. Usually, when the user attempts to interacts with a certain device, the associated functionality is to be utilized immediately without further downtime. For this reason, the memory allocation has been shifted to the point, where the device is created, i.e. the `as_control` device.

Once `open` has been called on the device, the allocated memory region can be mapped using the file operation `mmap`. For the `open` and `close` file operations, the default implementations provided by the kernel are used. For actually mapping the allocated memory to the user space, the `vm_end` and `vm_start` field of the provided *virtual memory aread (vma)* structure for `mmap` is used. The difference between both is used for determining the number of bytes to be mapped. If the number of bytes is equal or less than the one of the allocated area, the requested amount is mapped into the virtual address space of the user. Listing 4.3 lists the functions used for the actual mapping. The first one, `virt_to_pfn`, is a macro for determining the *page frame number (pfn)* of a given virtual address. Since __get_free_pages returns a virtual kernel address for the allocated memory area, this address is used. The page frame number is required by the following function remap_pfn_range, which performs the actual mapping. For the argument virt_addr, the vm_start field of the vma structure is used and for size the aforementioned difference of the vm_end and vm_start field. Similar, the vm_page_prot field is used for the prot argument. Usually, the user has to provide PROT_READ and PROT_WRITE when calling mmap from user space in order to be allowed to read and write to the mapped region. The kernel provides the virtual address to user after the mapping, without requiring the developer of the device driver to perform any additional tasks.

```c
#define virt_to_pfn(kaddr)

int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_addr, unsigned
unsigned long size, pgprot_t prot);
```

Listing 4.3: Functions for allowing to access physical addresses

Although the user can perform read and write accesses to the mapped memory area, transferring data to or from the hardware requires an explicit configuration of an appropriate memory module, depending on the desired direction. This task is accomplished by using the file operation ioctl in user space, which results in calling the unlocked_ioctl method of the as_mmap device by the kernel. For its arg parameter, the as_ioctl_params_t structure is used. The cmd field is used for determining the direction of the transfer, which can either be AS_IOCTL_CMD_READ or AS_IOCTL_CMD_WRITE. The commands are defined in the interface header file as_linux_kernel_if.h of the device driver. The former requires a as_memwriter module, whereas the latter requires a as_memreader module.

The following parameter address specifies the base address of the memory module to be used for the data transfer. Since the as_mmap device is not bound to a specific memory module, an appropriate memory module has to be associated, which is either a as_memreader, if the parameter AS_IOCTL_CMD_WRITE has been provided, or a as_memwriter for AS_IOCTL_CMD_READ. In order to find an appropriate memory module, the *device array* is iterated to check if a as_memio device exists, which is associated with the desired memory module. This way, the as_mmap device can use any of the defined memory modules within as_driver. If the memory module is defined within a as_memio device, it checks the busy flag of the device, whether it is currently in use, i.e. the file operation open has been called on the requested as_memio device. By checking the flag, the device driver prevents interfering with any ongoing data transfers or the *Buffer Handler*. The flag is a variable of the type *atomic_t* which also helps to prevent possible race conditions for acquiring the shared resource, i.e. the memory module.

The value parameter is used to specify the number of bytes to be transferred between hardware and software. Here, the device driver checks whether the requested amount can be transferred with the memory bus interface width used by the memory module. The number of bytes has to be a multiple of the interface width. Otherwise, a message is printed by the device driver to indicate the mismatch.

Lastly, the parameter user_addr_start is used to choose the start address within the memory mapped area, where the data transfer is supposed to begin. The address is desired starting point within the virtual address space mapped to the user and is translated into the associated physical memory address. Figure 4.5 illustrates the combination of the user_addr_start and value parameter for determining the memory area to be used for the data transfer. The sum of both parameters must not exceed the boundaries of the memory area, as the device driver will perform any data transfer.

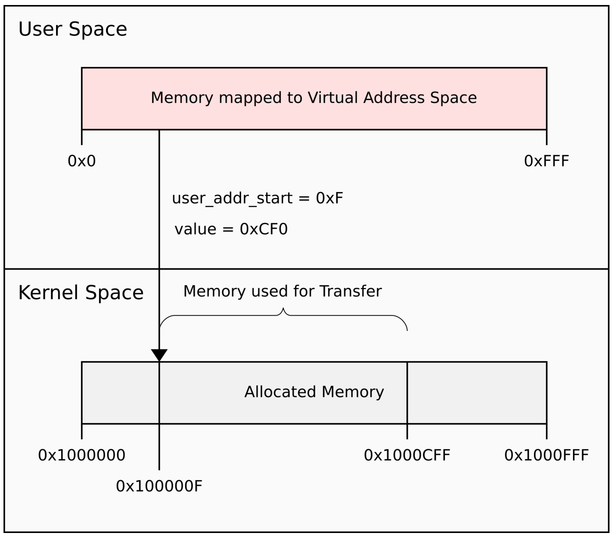

Figure 4.5.: Determining memory area for data transfer.

After successfully determining the physical address area for the data transfer and acquiring the corresponding memory module, the device driver configures the memory module accordingly. This is task is accomplished by using the interface functions of the `as_reader_writer` driver, which, in turn, uses the `as_regio` device for accessing the hardware registers. Thereby, the *size* and the *start address* of the *section* is written to the hardware register. For the remaining parameters, the default values specified in the `as_reader_writer` module driver are used. As of the `as_memwriter`, the `enable` and `disable_on_no_go` flags are set. This prevents the `as_memwriter` to store any additional data in its *Fifo Buffer* and thus negatively impacting other parts of the processing chain. Subsequently, regardless of the memory module being used, the `go` flag is set to start the operation of the module.

Since the memory module is occupied with a data transfer at this point, it cannot be released by the `unlocked_ioctl` method immediately, as the `as_memio` device performs a reset on the memory module upon `open`. This would terminate the current data transfer and thus has to be prevented. Therefore, the status flag `done` is checked for determining whether the memory module has already finished its operation. Depending on the image processing chain, this may take quite some time, which makes having to actively wait on the operation to finish undesirable. For this reason, the `as_memio` device utilizes a *wait queue*, which is initialized upon creating the `as_memio` device. After checking the `done` flag, the process executing the `unlocked_ioctl` method blocks by calling `wake_event_interruptible`. If the process is woken up by the *interrupt handler*, it checks the flag again. In case the flag is still not set, the procedure is repeated. Otherwise, the `busy` flag of the utilized `as_mmap` device is unset, thus releasing the memory module. Due to the aforementioned reason, the `as_mmap` device currently does not support the `O_NONBLOCK` flag.

4.5.8. Control Device

The `as_control` device is used for creating and deleting additional devices at runtime. It is the only device, which is created upon loading the device driver to the kernel and is always associated with the first *minor number*, i.e. "0". Similarly, it is the device with the first index within the *device array*. Since it is only used for managing the other devices of the driver, it only utilizes the mutex `access_lock` of its associated `device_data_t` structure, which is also initialized upon loading the device driver. For obvious reasons, `as_control` device cannot delete itself or add additional instances of this device type to the device driver.

The functionality of the `as_control` device is covered by the file operation method `unlocked_ioctl`, which is the only method within its file operation structure. As for all devices, `open` has to be called first, before being able to utilize it, however, no implementation is provided for neither `open` nor `close`. This results in using the default methods provided by the kernel.

The mutex `access_lock` is utilized to tackle potential race conditions. Therefore, it always locks its mutex first, when executing its `unlocked_ioctl` method. Since the `as_control` device is the only one being available after having loaded the *ASTERICS* device driver, adding further devices is the first task performed by the device and is therefore covered first. In addition to the pointer to the structure representing the device (filp), `unlocked_ioctl` method requires two parameters, namely `cmd` and `arg`. The first parameter is used tell the *as_control* device whether the call to its method stems from the user or kernel space. This is represented by providing either `CALLED_FROM_USER` or `CALLED_FROM_KERNEL`, respectively. Both defines are part of the header file `as_linux_kernel_if.h`, which is part of the *ASTERICS* device driver, but is also included for the application software. Depending on the origin of the call, the `arg` parameter has to be handled differently. For calls from kernel space, the data fields of the parameter can be accessed directly using common assignments. However, if the call originates from user space, the function `copy_from_user` has to be used for copying the data fields of `arg` into a local structure of the same type, in order to be able to access them. Generally, the request for creating or deleting devices is performed from user space, since the applications software determines the required devices for operating with the hardware residing on the FPGA. Nonetheless, the access from kernel space is also supported, in case a different device driver is used for requesting additional devices. As of the current state, this is mainly for future developments and applications of `as_driver`.

The `arg` parameter, as already suggested, contains the information required by the `as_control` device for creating devices. In order to retrieve the information in a more convenient manner, a structure is used, which posses a number of fields. This structure is of the type `as_ctrl_params_t`, which is part of `as_linux_kernel_if.h`.

The first field of this structure indicates, whether a new device has to be created, by providing `CMD_CREATE_DEVICE` for it. If this parameter has been provided, the `as_control` device checks, whether there are remaining entries in the *device array* for adding an additional device. This is done by comparing the global variable `initialized_devices`, used for tracking the number of currently present devices, with `MAX_DEVICES`, defining the maximal number of supported devices. Each time a new device is added to the device driver, this variable is incremented. Conveniently, this can also be used for index within the *device array* for inserting the next device. If the maximal number of devices has been reached, the `as_control` device exits with an error message and return value, pointing out this circumstance. Otherwise, the structure of the current device is initialized with default values, which mainly consists of setting pointers to NULL. In the next step, the type of the requested device is determined by evaluating the parameter `dev_type` of the `arg` parameter. The type has to be one of the supported ones, which are also defined in `as_linux_kernel_if.h`. Currently supported devices are the `as_regio`, `as_i2c`, `as_memio` and `as_mmap` device. The appropriate file operation structure is assigned to the `fops` field of the device structure. The `interface_width` and `flags`, for specifying the memory bus interface and supported direction for data transfer, are assigned to fields with the same name of the device structure, respectively. Subsequently, the `address_range_size` size field is assigned with the field with the same name. The device is then created by using `cdev_init` with the appropriate file operation structure.

For the `as_regio` and `as_i2c` device, a named memory region is requested. The fields `dev_address` and `address_range_size` of the `arg` parameter are used for specifying the physical start address and the size of the mapped region. The address of the mapped region is assigned to the `baseaddress_virt` field of the device structure. The offset between the physical and virtual address is stored in the field `offset`. Since the memory region has to have a name, `as_iic` and `as_regio` are used for the devices, respectively. However, as the names for the regions are unique, the device driver currently only supports one instance for each of the two device types.

For `as_memio` devices, only the field `dev_address` of the `arg` parameter is assigned to the field `hw_module_addr`, to associate the memory module. No further resources have to be allocated, since it is handled in its entirety within the file operation methods of the device. Since the device is the only one associated with a specific hardware module, the aforementioned field is only used for this device.

Regarding the `as_mmap` device, a new `mmap_info_t` structure is allocated for storing the `address_range_size` and the allocated physically concurrent memory, using *__get_free_pages*. The size is used for determining the one for the memory allocation. The start address of the allocated structure is assigned to the `mmap` field of the device structure. Lastly, the *wait queue* `wait` is initialized for allowing the device to block later on.

After the successful initialization of the device, it is published to the kernel by using `cdev_add`. For the *minor number*, the current value of `initialized_devices` is added to the first number, which has been used by the device driver. In this case, the first *minor number* is 0. Lastly, the `initialized_devices` variable is incremented and the mutex is unlocked again.

In order to prevent having to reload the kernel module of the *ASTERICS* device driver for removing devices, the `CMD_REMOVE_DEVICE` has been introduced to the `as_control` device. Since an array of a static size is used for managing the devices, deleting single devices would result in holes within the array, which would require a specific handling to avoid trying to access non-existing devices. Alternatively, an array with a generic size could be used, where the size is increased or decreased each time a new device is created or deleted, respectively. This approach would necessitate to allocate and reallocate major parts of the device representation at runtime. As resources may grow scarce the longer the operation system runs, trying to create new devices are more likely to fail. Usually, devices are deleted and replaced when the hardware processing chain is replaced. Often times, the majority of the devices are affected, which is the main reason for having the `as_control` device delete all other devices at once. This is accomplished by looping the *device array* and releasing all previously acquired resources. The variable `initialized_devices` is used as the upper boundary of the loop, since it represents the number of devices, which are currently used. After the `as_control` device has finished its task, `initialized_devices` is set to 1 again, as only one device is present at this time, i.e. the `as_control` device itself.

When deleting devices, care has to be taken, that none of these devices is currently used to prevent undesired behavior. For this reason, it is recommended to call `close` on all devices except the `as_control` device.

Figure 4.6 summarizes the essential functionality and the required steps of the `as_control` device.

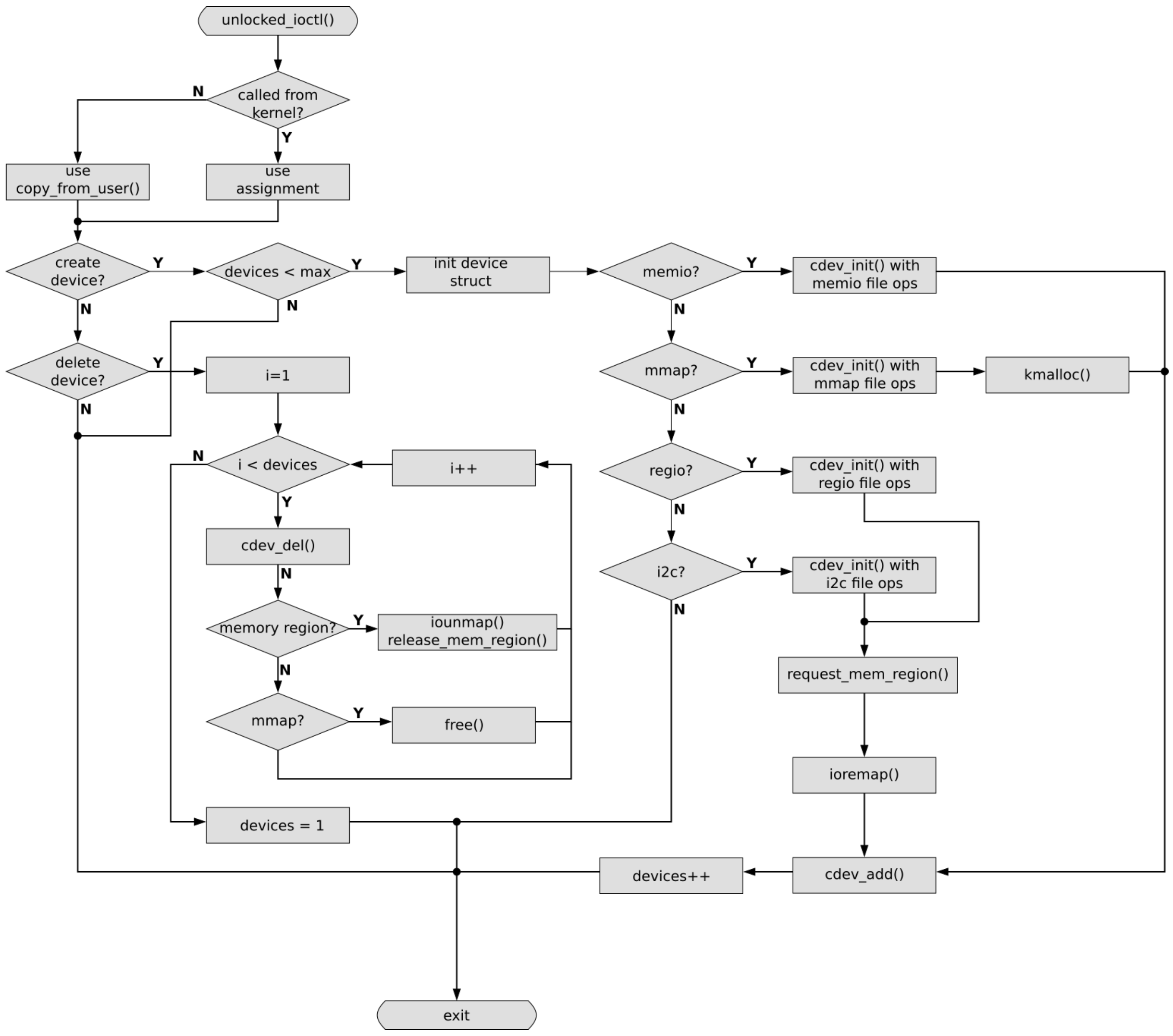

Figure 4.6.: Flow chart of the unlocked_ioctl file operation for the control device

4.5.9. Interrupts

4.5.9.1. Kernel Timer

For introducing interrupts to the `as_driver`, a global timer has been added for periodically generating interrupt events. The timer `interrupt_timer` is of the type `struct` timer_list, which is part of the timer API of the kernel. The timer uses jiffies for determining its interval between interrupt events. For the Zynq platform, the default setting of the system uses the value 100 for its HZ definition (param.h which is part of the system). This results in a granularity for the timer interval of 10 ms. The timer is initialized upon loading the device driver into the kernel, where the parameter TIMER_INTERVAL is used for the actual interval of the timer. For its interrupt handler, the timer is associated with the function timer_callback, which is called when the timer expires, i.e. after the configured interval. The timer is triggered for the first time at the end of the initialization of the device driver and registers itself within its interrupt routine. In order to prevent race conditions when the device driver is unloaded, the flag timer_shutdown is used. This flag is set exit method of the device driver before trying to delete the timer. The timer checks this flag before trying to register itself to run again. If this flag is set, the timer does not run again.

As aforementioned, the main purpose of the interrupt handler is to manage blocking as_memio and as_mmap devices. Since interrupt routines have to be executed in a timely manner, the amount of operations of it are rather limited. As as_driver may utilize several devices, serving them can be time consuming task and is therefore not suited to be performed within the interrupt handler itself. For this reason, the shared work queue by the kernel is used, into which the work item timer_wq, is inserted. This work item is associated with a function, which is executed when the work item is scheduled to run by the work queue. The function in turn is used for carrying out the actual operations required for handling the as_memio and as_mmap devices. The work item is set up and associated with the function data_transfer_update_task by calling the macro INIT_WORK. The function is then scheduled by the timer_callback function of the timer. This allows the interrupt handler of the timer to return in a timely manner, since the only two operations performed by it is calling schedule_work for the inserting the work item into the queue and registering itself to run again. The initialization of the work item is done just before configuring the timer, in order to be used.

The first part of the data_transfer_update_task function handles any as_memio devices, which are currently in use. In a first step, the function iterates all currently initialized devices within the *device array* to find as_memio devices. This is determined by examining the fops field of the device structure, whether it points to the file operation structure used for the as_memio device. If this is the case, it checks if the memio_active flag is set, which means the user has called open on the corresponding device. Subsequently, the register_intr flag has been set, if the device is currently sleeping. The data_transfer_update_task sets the condition variable wake_up_cond and calls wake_up_interruptible on the device, which causes it to wake up and check if new data has become available. For as_memio devices which are in use but the register_intr flag has not been set, the function as_memio_hw_update of the as_memio module driver is called. This guarantees, that data in the *Ring Buffer* is not "stuck" for an extended period of time, before being transferred. Otherwise the user may wait indefinitely for the data to actually appear on memory or at the processing chain on the FPGA. For the as_mmap devices, the fops field is also checked for determining the device. This is performed at the same time as for the as_memio devices, to prevent having to loop the devices twice. Similar to the as_memio device, the register_intr flag is examined before setting the wake_up_cond variable and calling wake_up_interruptible on the as_mmap device.

4.5.9.2. Hardware Module

`- This section is currently under construction -`

4.5.10. Driver Initialization and Deinitialization

Once `as_driver` has been successfully compiled, it can be loaded as a kernel module. The device driver requests a single major number and the configured amount of minor numbers, starting with 0. The major number is used to tell the kernel which driver is associated with a given device. Since a device driver may be responsible for more than one device, as is the case for the *ASTERICS* device driver, the minor number is used to distinguish between the devices. Currently, the device driver is assigned a major number dynamically by the operating system, instead of having it request a specific number itself. This prevents potential conflicts with other drivers, if the requested major number is already in use or a different driver tries to request the same major number at a later point. After the major number and minor numbers have been acquired, a device class is created for grouping the devices.

Since the devices depend on a specific *ASTERICS* -chain, they have to be created dynamically by the `as_control` device. For this reason, the device driver creates the `as_control` device upon initialization and assigns the first minor number (0) to it. The `as_control` device is created, by initializing the first device in the device list as the `as_control` device. Additionally, the number of initialized devices is incremented by one. However, the device driver does not create a device node on the file system itself, in order for the user to choose the name and location on his own. Rather, the user is expected to create the device node to the actual device.

Lastly, the interrupt timer is set up by mapping its service routine and configuring the time interval, at which the service routine is to be executed. The time interval can be configured as part of the compile-time options of `as_driver`.

As for unloading the *ASTERICS* device driver, the `timer_shutdown` flag is set to prevent the interrupt timer to run again before deleting it using `del_timer_sync`. Subsequently, all currently initialized devices, except for the `as_control` device, are deleted by releasing all resources to the operating system and removing any memory mappings, first. After this step, the device driver requests the kernel to delete the devices with their corresponding minor number. Once their is no device remaining, the `control` device is deleted in the same way, before the *device class* is destroyed and the driver itself is unregistered. The major number and all minor numbers are released and may be assigned to a different driver.

4.5.11. Application Notes

4.5.11.1. Building the Kernel Module

There are two options for building the Linux kernel module, either using a cross-compiler on the host platform or the native compiler on the target platform itself. Both require the proper *kernel headers* to be installed (usually at "/lib/modules") for building Linux kernel modules for the target architecture. A `Makefile` is provided for the `as_driver` with options for building the Linux kernel module on either the host or the target. Comments within the `Makefile` describe how either one can be used. The option for compiling on the host platform, however, has only been tested for Xilinx platforms so far. Xilinx ships ships its own libraries and tools, such as the compiler. Additional steps may be required for other vendors but should work in the same manner, by simply referring to the path where the kernel headers have been installed.

More information for installing the kernel headers are provided at the *Technical Wiki*.

4.5.11.2. Creating Devices

Except for `as_control`, all devices have to be created within `as_driver` since they depend on the actual *ASTERICS* -chain. Devices are defined within `as_hardware.c`, which provides interfaces for obtaining the number and list of devices. The source file `create-devices.c` shows the required steps for creating and deleting devices using the interfaces of `as_hardware.c(/h)`. This file can be found at ”tools/as-linux/src/kernel_module/create-devices” The shell script files `load_devices.sh` and `unload_devices.sh` use the executable of `create-devices.c`.

The whole directory (”create-devices”) can be copied into the one of the system for using the system specific `as_hardware.h/c`.

4.5.11.3. Creating Device Nodes

In order to use the devices of the *ASTERICS* driver, corresponding device nodes have to be created on the file system. The *minor* numbers are assigned in the same order as the devices are created, starting with ”1” (”0” is the `as_control` device). However, the *major* number of the kernel module has to be looked up, since it is dynamically assigned by the Linux kernel. This can be achieved by searching for `as_driver` in ”/proc/devices”. The actual device node is then created with *mknod*.

The two shell scripts `load_devices.sh` and `unload_devices.sh` are provided for creating and deleting device nodes. The *minor* numbers of the device nodes have to match the devices listed in `as_hardware.c`.

5. Interfaces

5.1. General Interfaces

by Alexander Zöllner, Gundolf Kiefer

5.1.1. Common Control and Status Registers

Note that the registers described here are currently only planned and will be integrated into the Automatics system generator, described in section 6.2, in the future.

Name	Address Off.	Width	Description
`asterics_id`	0x0	32	Indicates whether the processing chain is *ASTERICS* -based.
`asterics_version`	0x4	32	Major, minor and revision of the *ASTERICS* installation this chain has been built with.
`asterics_driver_id`	0x8	32	Compatible flavor of the software stack.
`asterics_state/control`	0xC	32	Global *ASTERICS* -chain instructions and status information.

Table 5.1.: Common control and status registers of an *ASTERICS* image processing chain.

Field Name	Bit Index	Type	Description
`asterics_id`	31:0	ro	The identification field of *ASTERICS* (for example: 0x0ee500a5).

Table 5.2.: Bit fields of the `asterics_id` register of the *ASTERICS* -chain.

Field Name	Bit Index	Type	Description
`major`	31:16	ro	Major number of the *ASTERICS* installation.

`minor`	15:8	ro	Minor number of the *ASTERICS* installation.
`revision`	7:0	ro	Revision of the *ASTERICS* installation.

Table 5.3.: Bit fields of the `asterics_version` register of the *ASTERICS* installation.

Field Name	Bit Index	Type	Description
`driver_id`	31:0	ro	Identification number of compatible software stack flavor. First 8 characters of the SHA1 hash of `as_hardware.c`.

Table 5.4.: Bit fields of the combined `asterics_driver_id` register of the *ASTERICS* -chain.

Field Name	Bit Index	Type	Reset Value	Description
`reset`	16	wo	0x0	Reset the entire *ASTERICS* -chain at once.
`sync_error`	3	ro	0x0	Data synchronization error occurred anywhere within the *ASTERICS* -chain which led to data loss.
`data_error`	2	ro	0x0	A data error occurred anywhere within the *ASTERICS* -chain.
`ready`	0	ro	n/a	*ASTERICS* -chain is ready for operation.

Table 5.5.: Bit field overview of the combined `asterics_state/control` register of the *ASTERICS* -chain.

The `READY` signal described in Table 5.5 is a ”AND” combination of the `READY` signals of all hardware modules. Similarly, the `DATA_ERROR` and `SYNC_ERROR` signal are ”OR” combinations of their respective signals of all hardware modules.

Usually, the hardware modules also poses a `state/control` register with the same bit fields as the `asterics_state/control` register.

5.1.2. Error Handling

There are two distinct error types available to *ASTERICS* , namely `SYNC_ERROR` and `DATA_ERROR`, which are handled separately. The former is a per-module signal, as described in Table 5.10 of Chapter 5.2, which may be forwarded to the `asterics_state/control` register of the *ASTERICS* -chain. The `SYNC_ERROR` cannot be cleared by software directly. Rather, a reset has to be performed on the *ASTERICS* -chain (see Chapter 5.2.2).

The `DATA_ERROR` is part of the `as_stream` interface (see Chapter 5.3). As opposed to the `SYNC_ERROR`, the software is responsible for clearing the `DATA_ERROR`, as described in Chapter 5.3.3.

5.1.3. Module and Chain Reset Behavior

Under certain circumstances, the need for reseting the *ASTERICS* image processing chain may arise, for example for having a defined state of the processing chain before using it for the first time or if an error occurred.

An *ASTERICS* -chain can either be reset at once using the `RESET` signal of the `asterics_control` register or per-module. The latter requires to perform the reset in the correct order, starting from the data source to sink.

If a module receives a `RESET` signal, the corresponding `READY` and `STROBE` signals have to be unset (0) after the following clock cycle. `READY` and `STROBE` must not be set (1) as long as `RESET` is set (1). The `RESET` signal has to persist for at least one clock cycle. Once the `RESET` is unset (0), the module sets its `READY` signal as soon as data can be accepted or the `STALL` signal can be set correctly.

The software is expected to read the `READY` signal of all modules or the one of the `asterics_state` register before starting any module.

5.1.4. Version Management

ASTERICS allows to build a wide variety of image processing chains, each being accompanied with their own ASTERICS *Support Package.* Thus, the ASTERICS *Support Package* has to match the configuration of the *ASTERICS* -chain, which includes address offsets or the type of modules which have been used. If the software does not match the hardware, unexpected behavior or the *ASTERICS* -chain not working at all is to be expected. The most common error is software drivers trying to access the module's corresponding hardware registers at a certain address but inadvertently configure registers of a different type due to mismatch of addresses.

For this reason, the *ASTERICS* -chain poses version registers as described in Chapter 5.1.1. The software can read these registers to check whether the current version of the software stack is compatible with the *ASTERICS* -chain it is trying to operate.

The ASTERICS *Support Package* also poses a version, which is a SHA1 hash of the source file `as_hardware.c`. This file describes the configuration of the *ASTERICS* -chain, along with pre-synthesis parameters of the used modules.

5.1.5. *i2c* Bus Master for Module Configuration

by Philip Manke

External modules used with the *ASTERICS* framework (particularly cameras) often require to be configured using the *i2c* bus. To provide a universal solution across platforms for this problem, *ASTERICS* implements its own *i2c* hardware module.

This section will give a brief overview on what *i2c* is and how to get started using the `as_iic` module.

5.1.5.1. What is *i2c*?

i2c is a bus system which allows for multiple devices to be connected using just two signals/wires in total. There are two types of devices which connect to the bus: masters and slaves. Only masters are allowed to start a transaction on the bus. Having multiple masters on a single bus is not supported by the `as_iic` module, though generally possible.

The two wires of the *i2c* bus carry a clock signal (SCL $\hat{=}$ Serial CLock) and a data signal (SDA $\hat{=}$ Serial DAta).

The *i2c* protocol uses device addresses to talk to the different devices on the bus. This address is 8 bits in length, with the last bit differentiating between read and write transactions.

5.1.5.2. How to Configure `as_iic` for Operation

The `as_iic` module only requires one function to be run before it is fully operational: `as_iic_init()`. This configures the SCL frequency of the *i2c* bus and is effective immediately. The supported standard *i2c* bus frequencies are 100 kHz and 400 kHz for *standard mode* and *fast mode i2c* respectively. With the `as_iic` module it is possible to configure the frequency freely in a range from 10 kHz to 1 MHz. This allows for faster transactions, provided the slave devices support the faster bus frequency. The function `as_iic_reset()` resets the module to the uninitialized state.

5.1.5.3. Using `as_iic`

The software driver for `as_iic` provides easy-to-use high level functions for executing transactions on the *i2c* bus. After connecting the hardware and initializing the `as_iic` module correctly, the high level driver functions can be used immediately.

The two most useful functions are:

- `as_iic_read_reg()`
- `as_iic_write_reg()`

These two functions will read and write the registers of *i2c* slaves connected to the bus. They both need the *i2c* modules' base-address, the *i2c* address of the slave and a pointer to the address of the register to read or write. The forth pointer required by the functions points to the byte to send to the *i2c* slave (`as_iic_write_reg()`) or to the variable to store the content of the *i2c* slave register in (`as_iic_read_reg()`).

Section 7.6.2.5 briefly describes how to connect hardware devices to the *i2c* bus.

Details on the software driver for `as_iic` and other modules, can be found in the *ASTERICS* Doxygen documentation. The implementation of `as_iic` is explained in detail in 7.6.

It is recommended to check out 7.6.4 before using `as_iic`, to avoid common pitfalls.

5.1.6. Generic Register Interface for *ASTERICS* Modules

by Philip Manke

ASTERICS provides a generic and configurable register interface that all modules can use. Configuring it becomes very easy when used in conjunction with Automatics, the *ASTERICS* chain generator. Table 5.6 lists the ports comprising the slave register interface from the viewpoint of an ASTERICS module.

Name	Data Type	Direction	Description
`slv_status_reg`	`slv_reg_data`	out	Data transport to software. An array of 32 bit registers.
`slv_ctrl_reg`	`slv_reg_data`	in	Data transport to hardware. An array of 32 bit registers.
`slv_reg_modify`	`std_logic_vector`	out	Data modify enable signals for the status registers. A vector as wide as the number of registers. Every clock cycle that a bit is set to `'1'` in this vector, the register will update it's value as set from the hardware.
`slv_reg_config`	`slv_reg_config_table`	out	A port to export the register configuration. An array of two bit `std_logic_vectors`, designating how the registers are configured.

Table 5.6.: Overview of the signals comprising the slave register interface of *ASTERICS* modules.

Each module using the interface must implement all four ports. The `helpers` package from the `asterics` library must also be used in the VHDL file. This makes the custom data types used to bundle the register signals available within the module.

Additionally, the module must contain a constant named `slave_register_configuration` towards the top of the architecture defintion:

```
constant slave_register_configuration : slv_reg_config_table(0 to
<reg_count>) := (<config>)
```

The width of the constant and all ports must manually be set to the number of the desired register count. Then the constant can be configured: For each register a two bit wide `std_logic_vector` is required, table 5.7 lists all possible values.

Value	Description
`AS_REG_NONE` or `"00"`	No register generated.
`AS_REG_STATUS` or `"01"`	Only generate a status register: Data transport only from the hardware module to the software. This saves some hardware resources and increases data security, as the software won't be able to overwrite the register contents.
`AS_REG_CONTROL` or `"10"`	Only generate a control register: Data transport only from software towards the hardware module. This mainly just saves some hardware resources.
`AS_REG_BOTH` or `"11"`	Generate a full slave register with data transport enabled in both directions.

Table 5.7.: Possible values in the slave register configuration constant.

Example configuration:

```
constant slave_register_configuration : slv_reg_config_table(0 to 3) :=
("11", "11", "10", "01");
```

For this configuration, all ports also need to have a data width of `(0 to 3)`. Also note that all register ports and the constant have data widths in an ascending direction (using `to` instead of `downto`)!

Note: You may use an ”others” construct to assign the register configuration (`:= ("11", (others => "01"))`). This is supported by Automatics for automatic IP-Core generation. *Important:* This causes an error when using the constant definitions `AS_REG_XX` and may thus only be used with string literals (eg. `"01"`)!

Note: If the module only has a single register, the ”normal” assignment of the configuration constant using `:= ("10")` will usually fail. Either use an `others` construct or a direct assignment using `:= (0 => "10")` to assign the register configuration. Both ways are supported by Automatics.

Using Automatics:

Automatics can automatically instantiate the required register manager for the register interface. This requires the following:

- The ports of the register interface must contain the names as mentioned above. They may have a *common suffix*, which is required if a module contains multiple interfaces. The suffix must be separated by an underscore from the specified signal names and the configuration constant must have the same suffix. Note: The suffix is only required if the module has multiple register interfaces.
- The configuration constant must be defined in the architecture definition, before any `begin` keyword, as Automatics stops parsing the architecture at the first `begin` keyword.

Manual configuration:

The register interface ports must be connected to the register manager component `as_regmgr`. The register manager has an interface consisting of the same ports with identical names as the register interface of the hardware modules:

- `slv_status_reg => slv_status_reg`
- `slv_ctrl_reg => slv_ctrl_reg`
- `slv_reg_modify => slv_reg_modify`
- `slv_reg_config => slv_reg_config`

And an interface roughly matching an AXI interface, ommiting unnecessary signals. Table 5.8 lists these ports.

Name	Data Type	Direction	Description
`sw_address`	`std_logic_vector`	in	The read and write address. `as_regmgr` does NOT support simultanious read and write accesses.
`sw_data_out`	`std_logic_vector`	out	Data transport towards software.
`sw_data_in`	`std_logic_vector`	in	Data transport towards hardware modules.
`sw_data_out_en`	`std_logic`	in	Enable signal for read accesses from software.
`sw_data_in_en`	`std_logic`	in	Enable signal for write accesses from software.
`sw_byte_mask`	`std_logic_vector`	in	Byte-wise mask for partial write operations. Fully supported by `as_regmgr`.

Table 5.8.: Overview of the signals comprising the ports connected to the AXI Slave manager of the *ASTERICS* slave register manager.

Integration details:
The software address is only as wide as the addressing of the hardware modules requires. Therefore the required bits have to be extracted from the address provided by the AXI Slave manager. The address will also have to be multiplexed from the read and write addresses.
The `sw_data_out` signal will also have to be multiplexed towards software (the AXI Slave manager).
Static HDL code has been developed and is used by Automatics to accomplish these tasks, shown in listing 5.1. Note that the constants have to be assigned values chosen for

the system that the code is used in.

```
  -- Register interface constants and signals:
  -- Register address width in as_regmgr
  constant c_slave_reg_addr_width : integer := 6;
  -- Module addressing width: ceil(log2(module count))
  constant c_module_addr_width: integer := 2;
  -- Register addressing width: ceil(log2(register count per module))
  constant c_reg_addr_width : integer := 4;
  -- Number of as_regmgrs
  constant c_reg_if_count : integer := 4;
  -- Which module / as_regmgr is addressed?
  signal read_module_addr : integer;
  -- Current address for as_regmgrs
  signal sw_address : std_logic_vector
                      (c_slave_reg_addr_width - 1 downto 0);
  -- Collect the sw_data_out signals of all as_regmgrs
  signal mod_read_data_arr : slv_reg_data(0 to c_reg_if_count - 1);

begin

  -- Extract the module address from the AXI read address
  read_module_addr <= to_integer(unsigned(
    axi_slv_reg_read_address(c_slave_reg_addr_width + 1
                              downto c_reg_addr_width + 2)));

  -- Connect the read data out port
  --     of the register manager of the addressed module
  read_data_mux : process(mod_read_data_arr, read_module_addr, reset_n)
  begin
      if reset_n = '0' then
          axi_slv_reg_read_data <= (others => '0');
      else
          if read_module_addr < c_reg_if_count
                and read_module_addr >= 0 then
              axi_slv_reg_read_data <=
                  mod_read_data_arr(read_module_addr);
          else
              axi_slv_reg_read_data <= (others => '0');
          end if;
      end if;
  end process;

  -- Select between read and write address of the AXI interface
  --   depending on the read/write enable bits
  -- The register managers can only handle a single
  --   read/write per clock cycle
  -- Write requests have priority
  sw_addr_mux:
  process(axi_slv_reg_write_address, axi_slv_reg_read_address,
            axi_slv_reg_write_enable, axi_slv_reg_read_enable)
  begin
    sw_address <= (others => '0');
    -- Disregarding lowest two bits
    --   to account for byte addressing on 32 bit registers
    if axi_slv_reg_write_enable = '1' then
        sw_address <= axi_slv_reg_write_address
                      (c_slave_reg_addr_width + 1 downto 2);
    elsif axi_slv_reg_read_enable = '1' then
        sw_address <= axi_slv_reg_read_address
                        (c_slave_reg_addr_width + 1 downto 2);
    else
        sw_address <= (others => '0');
    end if;
  end process;
```

Listing 5.1: Static code used to manage multiple `as_regmgr` modules

The register managers have five Generics, configuring data widths and which address they should listen for, listed in table 5.9.

Name	Data Type	Default Value	Description
`REG_ADDR_WIDTH`	`integer`	12	The register address width: ceil(log2(number of register managers)) + ceil(log2(number of registers per module)).
`REG_DATA_WIDTH`	`integer`	32	The register data width. Usually 32 bit.
`MODULE_ADDR_WIDTH`	`integer`	6	Number of bits used to address `as_regmgr` in this system.
`REG_COUNT`	`integer`	32	The number of registers for this `as_regmgr`.
`MODULE_BASEADDR`	`integer`	None	The address for this `as_regmgr`.

Table 5.9.: Overview of Generics of the *ASTERICS* slave register manager.

5.2. Common Per-Module Signals

by Gundolf Kiefer

5.2.1. Signal Overview

Table 5.10 shows the common signals that should be present in every *ASTERICS* module. Optional signals are marked with an asterisk (*). The column "Default" denotes the value that should be assumed for outer modules if the respective signal is absent.

All *ASTERICS* chains are synchronous designs the one common clock signal `CLK` with all flipflops being triggered with the raising edge of `CLK`. Unless noted otherwise, all other signals are active-high.

Signal	Direction	Default	Description
`RESET` (*)	in	0	Reset the module. This signal may only be omitted if the module is purely combinational. If set for 1 clock cycle, the module must reset itself.
`READY` (*)	out	1	Module is ready to operate. This is the response signal to RESET. If the reset process requires more than 1 clock cycle, the module may keep the READY signal low until the module is ready for normal operation. Once set (1), the module is not allowed to unset the signal unless a reset request was received. The READY signal may be omitted if the module is purely combinational or if the reset process is guaranteed to complete within one clock cycle.
`SYNC_ERROR` (*)	out	0	A Synchronization error occured. If the module encounters an error related to a potential loss of data or synchronization information, the module must raise this signal. At the same time, the module must go into a fail-safe state in which it does not generate any out potentially unexpected for subsequent modules. The SYNC_ERROR output set to 1 and the fail-safe behaviour must persist until the RESET input is set.
`FLUSH` (*)	in	1	Write out all internal buffers. All modules with internal buffers that do not flush automatically must implement this signal. If set, the module processes all internally buffered data, regardless whether new input data is arriving. If the signal is unset again while there is still data buffered, the module should, but is not required to stop processing.

Table 5.10.: Per-Module Signals ((*) = optional)

5.2.2. Module Reset and Error Handling

The signals RESET, READY, and SYNC_ERROR should all be made accessible to software (i. e. as I/O register bits and as part of the module driver).

On chain level, it is recommended to introduce global signals with the same names and make them accessible via I/O register bits. The global READY bit should be a logical ”and” of all module-level READY signals. The global SYNC_ERROR bit should be a logical ”or” of all module-level SYNC_ERROR signals.

The provision of a global RESET signal is particularly helpful to avoid potential errors due to an incorrect reset order if the modules are reset individually by software. If a global reset mechanism is missing, the software must reset the modules in a correct topological (half-)order, starting at the input modules and proceeding towards the output modules. Otherwise, a module that has just been reset may receive input data from a not-yet reset module and get into an unwanted state or produce undesired output.

5.2.3. Module Flushing

Modules may contain internal data buffers. For example, memory writer modules collect a number of data words in order to write them out efficiently using burst transfers. 2D Window Pipelines must buffer multiple lines of image data for their operation. If a frame-oriented processing chain is operated in a single-shot mode with potentially long periods of time without new incoming frame data, it can thus happen that data of a previous frame resides inside such buffers without being processed further. In consequence, follow-up (software) modules waiting for the completion of a frame cannot proceed. In the worst case, this results in a deadlock: The application software is waiting for the *ASTERICS* chain to deliver the last results of the current frame and would trigger a new frame afterwards. The *ASTERICS* chain, on the other hand, is inactive since now new data is coming in. This, however, does not happen, because the software is waiting.

The main purpose of the FLUSH signal is to circumvent such deadlock conditions. If set, the module is requested to completely process all of its internal data. In particular, if a unit of data (e.g. a frame) has been received completely at the input side, all output data of that frame must be emitted at its output side.

The FLUSH signal should be made available both to hardware (i. e. as an input signal) and to software (i. e. as a bit inside a slave register). In a typical use case, it is controlled by software.

5.3. The *ASTERICS* Streaming Interface (as_stream)

by Gundolf Kiefer

5.3.1. Signal Overview

An as_stream bus is used to transport mostly image, but also other types of data from one *source* module to one or multiple *sink* modules. The bus consists of an arbitrary number of data bits and a set of control signals.

The as_stream bus signals are listed in Table 5.11. Signals denoted with a right arrow (*rightarrow*) in column ”Direction” are outputs for the source module and inputs for the sink module(s). Accordingly, signals denoted with a left arrow (*leftarrow*) are directed from the sink module to the source module. Presently, only the `STALL` signal is directed backwards (from sink to source). If an `as_stream` bus connects to multiple sinks, an OR gate must be inserted to combine all their `STALL` outputs to feed the `STALL` input of the source module.

Optional signals are marked with an asterisk (*). The column ”Default” denotes the value that should be connected to the input signal of a module if the respective output is missing on the peer side.

Signal	**Direction**	**Default**	**Description**
`DATA<`n`>`	$\rightarrow$		User data. The width n and the data type are application-dependent, can be chosen arbitrarily with $n \geq 1$ and are not defined by the `as_stream` specification.
`STROBE`	$\rightarrow$		A new valid data item is present at the `DATA` bus.
`DATA_ERROR` (*)	$\rightarrow$		The present data item is invalid/unknown.
`STALL` (*)	$\leftarrow$	0	Request to pause sending data. This signal allows the sink module to put back pressure on its predecessor module if it is not able to process its incoming data in time. See Section 5.3.4 for detailed explanations on the mechanism.
`VSYNC` (*)	$\rightarrow$	–	Vertical (image) synchronization. This signal is optional. However, if it is present on the sink side, it must also be provided by the source. A clock cycle during which this signal is set marks the beginning (first pixel) of a new 2D image. Note: Some modules may operate incorrectly if this signal is set while `STROBE=0`.

Signal	Direction	Default	Description
`VCOMPLETE` (*)	→	`=VSYNC`	Image transfer completed. The signal indicates that an image has just been transferred completely and serves as a hint for subsequent modules. It should ideally be set during the clock cycle immediately following the cycle during which the last data item has been transferred.
`HSYNC` (*)	→	–	Horizontal (line) synchronization. This signal is optional. However, if it is present on the sink side, it must also be provided by the source. A clock cycle during which this signal is set marks the beginning (first pixel) of a new line. Note: Some modules may operate incorrectly if this signal is set while `STROBE=0`.
`HCOMPLETE` (*)	→	`=HSYNC`	Line transfer completed. The signal indicates that an image has just been transferred completely and serves as a hint for subsequent modules. It should ideally be set during the clock cycle immediately following the cycle during which the last data item has been transferred.

Table 5.11.: Signals of an `as_stream` bus ((*) = optional)

5.3.2. General Design Rules

5.3.2.1. Registered outputs

All output signals of an `as_stream` port have to be driven immediately by (a) a flipflop, or (b) an input signal, if there is absolutely no logic between this input and the output (not even a single gate!). Case (b) is particularly useful for the `STALL` signal to avoid long stall latencies requiring in potentially area-consuming buffers. In all other cases, it is safe and good to just always insert flipflops at the outputs.

5.3.3. Error handling

The `DATA_ERROR` signal indicates whether the value of a dedicated data item is incorrect or unreliable. Ideally, this information should be propagated through the whole chain up to the output, so that the application can read which parts of the data are correct and which are not.

In some cases, the `DATA_ERROR` information cannot be propagated up to the end. Examples are:

- The chain may end with an `as_memwriter` module, and the output data format has no fields to indicate the validity of individual pixels / data units.
- There are modules in the chain that do not support the `DATA_ERROR` signal.

In such cases, it is recommended to add a global status register bit `GLOB_DATA_ERROR`, which is set to one if the last element of a chain propagating the data error information raises its `DATA_ERROR` output and must be reset by software. This way, the application software can at least determine if some data units may be reliable between two polls of this status bit.

5.3.4. Stall Mechanism

The `STALL` signal enables modules which cannot guarantee to process a new input data word in each clock cycle to send a break signal towards their predecessor modules and slow them down just enough to let the whole chain operate correctly and at optimum speed.

Unlike all other signals, which are all directed forward and together allow build well-formed, very long pipelines without encountering serious physical problems, the `STALL` signal is directed backwards. To avoid over-long signal paths, it may need to be buffered and may thus require a certain (system- or chain-dependent) number of clock cycles until it is handed over from the overloaded module to the first, data-generating module of a complex chain. This must be taken into account by the chain designer, and buffer registers or FIFOs must be inserted into the chain to avoid data loss due to data latencies.

To help the chain designer in this task and to allow an automatic insertion of such buffers in the future, the following rules apply:

1. `STROBE` has precedence over `STALL`. A `STROBE` signal set to 1 strictly and unequivocally defines that there the `DATA` signal carries a new data item *now*, which will never be repeated. The sink module must take the data or raise a `SYNC_ERROR` condition, even if it has already set its `STALL` signal. In other words, `STALL` can be seen as a request or a hint, which may or may not be followed. Any module that cannot take over a new data item each clock cycle must be designed to deal with such a situation and do a correct `SYNC_ERROR` error handling.

2. An `as_stream` *source* port can be *stall-absorbing* or not. Whether or not a port is *stall-absorbing* should be indicated by the presence of the optional `STALL` signal, which should only be present for *stall-absorbing* modules. Exceptions must clearly be identified in the module documentation.

3. An `as_stream` *sink* port can be *stall-generating* or not. Whether or not a port is *stall-generating* should be indicated by the presence of the optional `STALL` signal, which should only be present for *stall-generating* modules. Exceptions must clearly be identified in the module documentation.

4. Certain filter modules which do not need to generate stalls themselves and cannot buffer much data internally may still implement the `STALL` port signals and simply propagate the `STALL` signal from their outgoing to their incoming `as_stream` port. Such modules are referred to as *stall-propagating* modules. Towards their successor, they act as a *stall-absorber*. Towards their predecessor, they act as a *stall-generator*.

5. Any sink port must accept the last data unit during the clock cycle in which the `STALL` signal is raised without errors.

6. A source port which is declared to be *stall absorbing* must not set its `STROBE` signal one clock cycle after which `STALL` has been raised.

In order to fullfil the condition 5, modules which forward their `STALL` signal from an output to an input must do this within the same clock cycle, i. e. by pure combinational logic without and flipflops. This may lead to overlong combinational paths during synthesis. Timing issues can be overcome by inserting dedicated `as_stall_buffer` modules into overlong paths, which insert a flipflop into the `STALL` line and a buffering register for all other lines.

5.4. The *ASTERICS* 2D Window Filter Interface (`as_window`)

by Alexander Zöllner

5.4.1. Signal Overview

A `2D window filter` is installed at a fixed position on a 2D-sliding window buffer [**pohl˙efficient˙2014**], which moves its data continuously forward similar to a conveyor belt. The `2D window filter` extracts data passing its input fields and processes the data. Results may be passed to a 2D-sliding window buffer again, as input for the next filter, which exhibits the behavior of an assembly line.

Signal	Direction	Description
`<`x`>` $\times$ `WINDOW<`n`><`m`>`	$\rightarrow$	Data from 2D-sliding window buffer. The dimension of n and m as well as the data type are application-dependent, can be chosen arbitrarily with $n, m \geq 1$ and are not defined by the `2D Window Filter Interface` specification. Additionally, the number of windows can also be chosen arbitrarily with $x \geq 1$.

Signal	Direction	Description
`DATA<`n`>`	$\leftarrow$	Data to 2D-sliding window buffer or different module. The width n and the data type are application-dependent, can be chosen arbitrarily with $n \geq 1$ and are not defined by the `2D Window Filter Interface` specification.
`STROBE`	$\rightarrow$	A new valid data item is present at the `WINDOW` and `DATA` bus.

Table 5.12.: Signals of an `2D window filter` bus

6. Tools

6.1. General Tools

`- This section is currently under construction -`

6.2. Automatics - The Chain Generator

by Philip Manke

This chapter introduces and describes the functionality of Automatics, the *ASTERICS* processing chain generator.

Automatics uses a short, user-edited Python script to generat, several possible output products, including hardware source files, software source files, a functional IP-Core (currently only in the Vivado-format) and a SVG graph.

Section 6.2.1 gives a brief overview of the functionality of Automatics, enough to get you going. For more details and a deeper understanding of Automatics, read sections 6.2.2, 6.2.5, 6.2.8, 6.2.3 and partly 6.2.6 and 6.2.7. If you want to use your own hardware modules with Automatics, take a close look at section 6.2.5. Sections 6.2.4 and 6.2.7 provide additional information about the internal structure of Automatics and more advanced configuration methods, which is generally not of interest to users, rather then developers of the *ASTERICS* framework. However, when attempting to describe highly complex systems or debugging errors, reading this section may proof useful.

6.2.1. User Guide

This section gives a summarized explanation of how to generate an *ASTERICS* system using Automatics.

Prerequisites:

- Python 3.5 or higher and (very) basic knowledge of Python syntax (you will need to edit a short script)
- An *ASTERICS* installation (clone or snapshot of the Git repository)
- (mostly optional) Basic knowledge of VHDL syntax
- (optional) GraphViz tool and python module `graphviz` for visualization
- (optional) A synthesis tool for your type of FPGA
- (optional) A hardware target - we suggest the ZyboBoard with an OmniiVision OV7670 camera module

This guide will walk through the generation of a demo system as included with ASTERICS. All necessary steps are explained, with a focus on steps involving Automatics.

Step 1: Setup your work environment

1. Locate your ASTERICS installation and move there on the command line.
2. If you have Vivado installed on your system and want to use it for this guide, source your Vivado settings file now. In the Xilinx installation directory, it should be located under `"Xilinx/Vivado/<version>/settings64.sh"`. ASTERICS requires the environment variable `XILINX_VIVADO` to be set to start Vivado automatically.
3. Source or run the script `"settings.sh"` in the root of the ASTERICS installation.
4. Navigate to the folder `"<asterics>/systems/"` and copy the folder `"as_refdesign_zynq"` to the location where you want to generate the system. Now you can start using Automatics.

Step 2: Modify the chain description (optional)
Move into the directory `"as_refdesign_zynq/asterics/image_differencing/"`. Here you will find the file `asterics-gen.py`, the chain description file.

An *ASTERICS* chain is comprised of modules that execute specific image processing and general data management tasks. You can add modules using the `chain.add_module()` method. This method takes two parameters: `chain.add_module("module name", "custom name")`. The module name is identical to the name of the VHDL entity of the module's toplevel VHDL file. The optional `custom name` parameter can be anything you want, as long as there are no duplicates in your system. This name is used to associate signal, port and address names to the module. To find out which modules are available, you can use the interactive Automatics environment or GUI by calling `as-module-browser-cli` or `as-module-browser` on the console, after sourcing the *ASTERICS* settings file. Alternatively you can browse the `modules` folder of your *ASTERICS* installation, the names of the contained folders are usually identical to the VHDL entities (there are exceptions and folders containing more than one module).

The `add_module()` method returns a reference to the newly created module object, which you should assign to a new variable. With this object, you can use a `connect()` method to connect two modules to each other. Several objects within Automatics provide `connect()` methods:

- *Module objects:* `<source module>.connect(<target>)`
  Example: `camera.connect(collect)`
- *Interface objects:* `<source>.connect(<target>)`
  Example: `splitter.get("1").connect(invert)`
- *Port objects:* `<source>.connect(<target>)`
  Example: `camera.get_port("vcomplete_out").connect(collect)`
- The `chain` object: `chain.connect(<source>, <target>)`
  Example: `chain.connect(camera, collect)`

The target used in a `connect()` method does not need to be of the same ”type” as the source. The data flow direction is assumed to be from the source to the target of the connect method call. Note: Before using an object as a target in a `connect()` method, make sure to have created and added it to the chain using the `chain.add_module()` method.

Using the `module.get("name", "direction")` method or the more specific `module.get_interface("interface name")` and `module.get_port("port name")` methods, you can tell `connect()` methods to connect individual ports and interfaces. Additionally, you can use the module objects returned by the `chain.add_module()` method to configure many aspects of the modules and how they are integrated into your *ASTERICS* system. For a reference of the available methods, read section 6.2.6.

Here are two examples for system definition functions:

Example 1 (listing 6.1): A simple processing chain, reading an image from an Omniivision camera in grayscale, inverting all pixels and writing the result into memory. This system uses just four hardware modules.

```

# as_invert demo system:
# (HW ->) camera -> invert -> collect -> writer (-> RAM)

# -------- Module instantiations ------------------------
camera = chain.add_module("as_sensor_ov7670", "camera0")
invert = chain.add_module("as_invert")
collect = chain.add_module("as_collect")
writer = chain.add_module("as_memwriter", "writer")

# -------- Module configurations ------------------------
camera.add_iic_master("XILINX_PL_IIC")
writer.set_generic_value("MEMORY_DATA_WIDTH", 32)
writer.set_generic_value("DIN_WIDTH", 32)

# -------- Module connections ---------------------------
camera.connect(invert)
invert.connect(collect)
collect.connect(writer)
```

Listing 6.1: Definition of a simple invert system

Example 2 (listing 6.2): A processing chain with multiple branches calculating the differences in pixel values of two subsequent video frames. This is the description for the example system you are building by following this guide.

```

# Difference demo system:
# (HW ->) camera -> splitter --[0]> ...
#                         \-----[1]> ...
# (RAM ->) reader -> disperse -[2]> ...
#
# -[0]-> collect0 -> writer0 (-> RAM)
# -[1]-> sync => pixel_diff -> collect1 -> writer (-> RAM)
# -[2]--->/


# -------- Module instantiations ------------------------
# Camera
camera = chain.add_module("as_sensor_ov7670", "camera")
camera.add_iic_master("XILINX_PL_IIC")
# Splitter
splitter = chain.add_module("as_stream_splitter")
# Reader
reader = chain.add_module("as_memreader")
# Disperse
disperse = chain.add_module("as_disperse")
# Stream Sync
sync = chain.add_module("as_stream_sync")
sync.set_generic_value("BUFF_DEPTH", 1024)
# Pixel Diff
diff = chain.add_module("as_pixel_diff")
# Collect modules
collect0 = chain.add_module("as_collect", "collect0")
collect1 = chain.add_module("as_collect", "collect1")
# Writer 0
writer0 = chain.add_module("as_memwriter", "writer0")
writer0.set_generic("MEMORY_DATA_WIDTH", 32)
writer0.set_generic("DIN_WIDTH", 32)
# Writer 1
writer1 = chain.add_module("as_memwriter", "writer1")
writer1.set_generic("MEMORY_DATA_WIDTH", 32)
writer1.set_generic("DIN_WIDTH", 32)

# -------- Module connections ---------------------------
chain.connect(camera, splitter)

# original image path:
chain.connect(splitter.get("0"), collect0)
chain.connect(collect0, writer0)

# previous image read path:
chain.connect(reader, disperse)

# sync image pixels, diff and write back the result:
chain.connect(disperse, sync.get("0", "in"))
chain.connect(splitter.get("1"), sync.get("1", "in"))
chain.connect(sync, diff)
chain.connect(diff, collect1)
chain.connect(collect1, writer1)
```

Listing 6.2: Definition of a system calculating pixel value differences over time

In the examples above, methods for configuring the hardware modules were used, that haven't been mentioned yet, like `module.set_generic_value()`. Section 6.2.6 gives an overview of all configuration methods, however, for brevity, here is a brief list of the most important methods:

- *module*.`make_port_external("port name")`: Automatics will pull this port to toplevel, so it faces the outside of the generated hardware
- *module*.`make_interface_external("interface name")`: Same functionality as "`make_port_external`" but for an entire interface.

- *module*.`set_port_fixed_value("port name", "value")`: Set this port to the specified value. Note: Automatics directly writes what you pass to the parameter `value` into the VHDL code, so make sure it doesn't violate VHDL syntax.
- *module*.`set_generic_value("generic name", "value")`: Same as the above command, just for VHDL Generics instead of ports. Generics are usually used to configure the functionality of hardware modules, so you will be setting things like image resolution, data bus widths, optional functionalities and the like.
- *module*.`make_generic_external("generic name")`: Automatics will propagate the generic to the toplevel - the interface of the generated hardware / IP-Core. This allows you to adjust the value of the generic using your synthesis tool of choice.
- *module*.`get_interface("interface name", "direction")`: Return the interface "interface name" of *module*. Use in conjunction with a `connect()` method to connect specific interfaces.
- *module*.`get_port("port name")`: Return the port `"port name"` of *module*. Use in conjunction with a `connect()` method to connect specific ports.
- *module*.`get("name", "direction")`: Return the interface or port `"name"` of *module*. This convenience method first calls `get_interface()` and, if no interface `"name"` was found, then calls `get_port()`. Note that the parameter `"direction"` only applies to interfaces. Use this method as a shorthand for either of the more explicit methods.

You may edit the system described in `asterics-gen.py`, note however, that the supplied software in the file `asterics-demo.c` might not be compatible with your changes. We suggest you first generate the system without having made changes to verify that the generator works as expected on your system. Afterwards you can copy the script and software - or the entire project - and use it as a basis for your own ideas.

Step 3: Generate the system
In the root of the demo system `as_refdesign_zynq` a Makefile is provided. You may use the Makefile to automatically generate the entire system, implement it using Vivado, compile the software and flash it to hardware. If you have a ZyboBoard and OV7670 camera handy, you can obtain a functional system by connecting the board to you computer and calling `make` on the commandline, provided your Vivado installation already includes the board files for the ZyboBoard.

Optionally, with Vivado installed you may generate the bitfile and software by running `make asterics_vivado_cores && make build_system` on the console. The output will be generated to the folders `hardware`, `vivado_cores` and `software`.

You can also just run the generator by executing the `asterics-gen.py` script using Python 3. In the commandline run:
```
python3 asterics/image_differencing/asterics-gen.py vivado vivado_cores
```
or using the Makefile: `make asterics_vivado_cores`
This will generate the ASTERICS IP-Core to `vivado_cores`.

If you don't have Vivado installed, you can skip the IP-Core packaging step to just obtain the source files, both hardware and software for the ASTERICS system by running:

`python3 asterics/image_differencing/asterics-gen.py core asterics_core`
or using the Makefile: `make asterics_core`
The output will be generated to `asterics_core`.
For all generator targets of this demo system, a SVG graph of the internal processing chain will be generated to the root folder of the system with the name `asterics_system_graph.svg`. In the Automatics script, this is achieved by the following command: `asterics.write_system_graph(chain)`

For more information on the demo system, read section 9.1.

For future IP-Cores, consider copying the script `asterics-gen.py` or the more generic script in `<asterics>/tools/as-automatics/user_script.py` to use as a template.

6.2.2. Default Behaviour

This section briefly explains the default behaviour of Automatics when handling Generics, Interfaces and Ports of the modules you add to a new ASTERICS system.

Generics you leave unconfigured will use the default value which is usually present in the VHDL code. If no default value is available, it will be left blank, causing an error during synthesis. Therefore you should check which default values are set for each generic, using the `module_detail()` function in the interactive mode or the GUI, described in section 6.2.8. Alternatively you can use `module.make_generic_external()` to enable editing of the Generic value using the synthesis tool of you choice or set a fixed value using `module.set_generic_value()`

Ports that are left unconnected will be handled individually. After all connection methods from the user script are handled and all standard ports, such as `clk` and `reset`, are connected, the modules are scanned for unconnected ports. For each port, their ruleset is scanned for applicable conditions, at this point, the `sink_missing` condition usually applies for unconnected ports and the associated action is executed. With the default ruleset in place, this means an info message is printed to the console and the port is left unconnected. During the code generation process, ports that are still without a connection target are set to their *neutral value*. For data outputs that means `open`, no connection. For data inputs that means `'0'` or `(others => '0')` if it's a vector data type. If you don't see any info messages about unconnected ports, make sure that the loglevel for the console is set to at least "INFO" using the `asterics.set_loglevel()` method, or check the logfile `automatics.log`.

Unconnected *Interfaces* are handled much the same as unconnected ports. Their associated ports are added to the list of unconnected ports and are handled in the same step as unconnected ports which are not part of interfaces. Some Interfaces may still be automatically connected if a matching interface template exists within Automatics (see the file `as_automatics_templates.py`) and the template calls for the interface to be connected externally. This is indicated by the boolean attribute `interface.to_external`, which can be set using the method `interface.make_external()`.

Automatics currently does not support connecting ports or interfaces to multiple sinks. This means that, in case your chain description contains multiple connect statements to the same sink, only the first connect statement will complete successfully, all subsequent statements will effectively be ignored.

6.2.3. Quirks of Automatics

This section describes some possibly quirky, confusing or unexpected behaviour of Automatics.

- Handling of the tilde character (∼): Automatics can not resolve the tilde character at the moment. It will be handled as a regular character part of the path, making it potentially annoying to clean up after.
- Parsing VHDL of custom or new modules: As the VHDL analyzer is relatively limited, not all valid VHDL syntax is parsed correctly. See section 6.2.5.1 for a detailed account.
- Connecting a port to multiple targets: At the moment, Automatics does not support this featur. Although it does not generate any error messages, only the first declared target will be connected. For connecting interfaces, use a splitter module.
- Multiple automatically managed slave register interfaces in a module: When implementing more than one slave register interface (see section 5.1.6) in a module managed by Automatics, the interface ports must be differentiated from the other interface(-s) using *only* a suffix. Otherwise some generated singals will have identical names, causing errors later on.

6.2.4. Automatics Objects

This section explains the basic functionality of each major component / Python class in Automatics. For most classes the most important attributes and / or methods will be listed.

Note: All paths you provide in any Python scripts can be either relative or absolute paths. The tilde character (∼) is *not* interpreted as the home directory by Automatics and must not be used as such.

6.2.4.1. asterics

The Python module `asterics.py` is used as a simple entry-point for all functionality of Automatics and is used to source all of Automatics in a single Python import.

Functions of the `asterics.py` wrapper module:

- `asterics.new_chain()`: Instantiate a new processing chain object (`AsProcessingChain`) to build a new IP-Core.
- `asterics.vears(folder, use_symlinks, force)`: Link or copy the VEARS IP-Core to *folder*. Use `use_symlinks` to choose between copying or linking. Default is `True` = linking. Use the `force` parameter to allow Automatics to delete the target file or folder if it already exists.
- `asterics.add_module_repository(folder)`: Have Automatics scan the contents of `folder` for *ASTERICS* modules and add found modules to the library for use in Automatics.

- `asterics.set_ipcore_name(name, description)`: Set the name and optionally the description displayed with the generated *ASTERICS* IP-Core.
- `asterics.define_hardware_target(partname, design_name, board)`: Set the target FPGA part name, design name and board model name used when packaging the *ASTERICS* IP-Core. Caution! The internal values from your synthesis tool must be used here, as you would when setting up a FPGA project.
- `asterics.set_loglevel(console, logfile)`: Set the level of messages output by Automatics to the console and the logfile. Available levels in ascending order of severity are: "debug", "info", "warning", "error" and "critical". Two shorthand methods are available for setting the "info" level (`asterics.verbose()`) and the "critical" level (`asterics.quiet()`).

6.2.4.2. Automatics

This is the main class that implements Automatics and controls all major processes used to generate an IP-Core. In the user script, Automatics is started by instantiating the `Automatics` class during the statement `import asterics`.

The class is indirectly accessed via the `AsProcessingChain` object provided by the `asterics.py` module and the `asterics.py` wrapper module itself. Thus, this class does not usually need to be used directly.

6.2.4.3. Port

Port is the most basic Python class that Automatics uses to represent an image processing system internally. As a user it is useful to know the most important attributes of the Port class and how they are used by Automatics.

Important attributes of Automatics class Port:

- `name`: This string attribute defines the Ports *abstract* name. It is not identical to the Port's name in VHDL code. For example: The port `strobe_in`, part of some interface, will have a `Port.name` attribute with the value `strobe`, without the suffix. The abstract names are defined in Interface templates, mentioned later.
- `code_name`: This string attribute represents the Port's actual name in VHDL code.
- `direction`: This string attribute describes the direction of data flow of this Port. Possible values are "in" and "out". "inout" is only supported for ports that are made external, such as for IIC or camera interfaces.
- `port_type`: This string attribute contains some meta information about the Port object. Valid values are: `"single"` - a normal port, `"external"` - the port will be connected to the IP-Core toplevel, `"interface"` - the port is part of an interface and `"register"` - the port is part of a special slave register interface.
- `data_type`: This string attribute contains the data type descriptor as it is defined in VHDL code and is used to determine compatibility between Ports.

- `data_width`: This tuple describes the data width for Ports with vector data types. It contains three items: A start expression, a separator (either `to` or `downto`) and an end expression - e.g. (7, ”downto”, 0). For non-vector data types it is always (1, None, None).
- `ruleset`: This list of rules describes, for the most part, how Automatics will handle a Port object during system generation. Details are described below.
- `optional`: This boolean attribute can mark Ports as optional. Automatics will omit most warnings and errors for optional Ports.
- `parent`: This attribute stores the next higher object in the hierarchy. This is either an Interface object or an AsModule object.

Port rules:

The Port class contains a namedtuple `Rule`, that is used to store rules in Port objects. Rules are stored as `condition, action`.

Here is a list of port rule conditions:

- `single_port`: If the Port's port_type attribute is `single`
- `external_port`: If the Port's port_type attribute is `external`
- `type_signal`: If the Port's port_type attribute is `signal`

All other conditions are applied when two modules or interfaces are connected to each other. Automatics tries to find a matching data source port for each data sink and vice versa.

Condition	Source Port	Sink Port	Comment
`both_present`	Found	Found	Default for connecting ports
`source_present`	Found	Don't care	Always true
`sink_present`	Don't care	Found	Usually the same as `both_present`
`sink_missing`	Found	Missing	Useful to define default values

Table 6.1.: Port rule conditions

Here is a list of port rule actions that can be triggered by the conditions:

Action	Description	Applicable to Condition
`connect`	Connect sink and source port to each other, after checking their compatibility. This includes a check of data type, data direction and name.	`both_present`

Action	**Description**	**Applicable to Condition**
`forceconnect`	Connect sink and source port to each other, after a reduced of checks. Only mandatory checks of data direction and data type are run.	`both_present`
`make_external`	Make the port available outside the IP-Core.	Any
`set_value(<value>)`	Set the port to the fixed value `<value>`, which is user editable.	`sink_missing`
`fallback_port (<port name>)`	Search for port `<port name>` in the or interface module the port should be connected to, and connect to it, if present.	`sink_missing`
`bundle_and`	Add this port to an and-gate and make the result external. Ports with the same name with this rule set will be bundled together. Useful for example when used with `ready` or `error` ports.	Any
`bundle_or`	Same as `bundle_and` but with an or-gate.	Any
`error`	Stop generation of the IP-Core and cite this rule as the cause. Includes information of the port and module that caused the action.	Any
`warning`	Print a warning message and cite this rule as the cause. Includes information of the port and module that caused the action.	Any
`note`	Print an info message and cite this rule as the cause. Includes information of the port and module that caused the action.	Any

Table 6.2.: Port rule conditions

For information on how to edit the ruleset of a Port object, read section 6.2.6.

6.2.4.4. Interface

The Interface class is an abstraction layer above Port. An Interface contains a list of Ports and usually an Interface template. Using the template, Automatics identifies interfaces by matching the Ports' abstract names, including pre- and suffixes, data type and data direction.

You can also define your own Interface templates to have them automatically recognized and connected by Automatics. Here is an example Interface template definition using *ASTERICS* `as_stream`:

```python
class AsStream(Interface):
    """Template definition for ASTERICS' 'as_stream' interface."""

    def __init__(self):
        # Interface name
        super().__init__("as_stream")
        self.add_port(Port("strobe"))
        self.add_port(Port("data", data_type="std_logic_vector",
                           data_width=Port.DataWidth("DATA_WIDTH - 1",
                                                     "downto", 0)))
        self.add_port(Port("data_error", optional=True))
        self.add_port(Port("stall", direction="out", optional=True))
        self.add_port(Port("vsync", optional=True))
        vcomplete = Port("vcomplete", optional=True)
        vcomplete.add_rule("sink_missing", "fallback_port(vsync)", False)
        self.add_port(vcomplete)
        self.add_port(Port("hsync", optional=True))
```

Listing 6.3: Definition of the as_stream Interface template

Other examples for template definitions can be found in Automatics source module `as_automatics_templates.py`. If you want to include a general new Interface template, we suggest using a separate script file that you import in your user scripts (*before* the `chain` object is created!). To add a template to Automatics globally use the command:
`AsModule.add_global_interface_template(MyInterface())`
Note that you need to import the `AsModule` class from the file `asterics/tools/as-automatics/as_automatics_module.py`
If you want to add an interface only to select modules, use:
`module.add_local_interface_template(MyInterface())`
in the module specification script before the call to `module.discover_module(...)`. Note that the template script has to be imported in each module specification script using it. For more information on the module specification scripts, read section 6.2.5.

Important attributes of Automatics class Interface:

- `name`: The interface name given by Automatics during the import of the interface's module. This name is later updated after the system is connected internally.
- `name_prefix`, `name_suffix`: Prefix and suffix strings that apply to all ports of the interface.
- `unique_name`: A unique name given to every interface automatically, so they can be securely identified during the build process.

- `type`: The interface type as defined by the template via the `super().__init__("type")` method. Identifies the interface type, e.g. `as_stream`
- `direction`: The abstract direction of the Interface. The direction of interface templates is always "in". The direction is only swapped if all ports have the inverse direction to that of the template.
- `to_external`: Boolean attribute; Whether this interface will be made external to the IP-Core.
- `instantiate_in_top`: Tuple attribute. Automatics will automatically instantiate the module specified in this attribute to the specified module group; tuple: ("module name", "module group"). The first element is the entity name of the module to instantiate, the second the name of the module group where the module should be instantiated. If no group name is specified, the module will be instantiated in the toplevel (`asterics.vhd`). For example: This attribute is used in the InternalMemoryInterface template that the `as_reader_writer` modules use to connect to the AXI bus. The required AXI Master controller is automatically instantiated in toplevel.
- `ports`: This is the list of all Port objects part of this Interface.
- `generics`: This is the list of all Generic objects associated to the Interface's Ports
- `template`: The Interface template that was used to automatically group this Interface. Not all Interfaces have this attribute set.
- `parent`: The module object this Interface belongs to.

For information on how to configure interfaces of modules in the user script, read section 6.2.6.

6.2.4.5. Generic

Generics are relatively basic objects, representing VHDL generics.

Important attributes of Automatics class Generic:

- `name`: Abstract name of the Generic.
- `code_name`: Name as written in the VHDL code.
- `default_value`: Default value of this Generic, as read in through the VHDL code.
- `data_type`: The Generic's data type.
- `value`: The Generic's value as set by Automatics or the in the user script. Can also be a reference to another Generic object.
- `to_external`: Default behaviour: Make the Generic adjustible through the IP-Cores toplevel VHDL file.

- `link_to`: Find a Generic on higher levels than this Generic to link to (to take on it's value). For example: The user added a Generic to the toplevel that set the system resolution. By setting `link_to` to that Generics name, Automatics will automatically set this Generic to the user Generics value.
- `value_check_function`: A Python function that runs when the Generic is assigned a new value. Can be used to generate errors in case an unsupported value is set.
- `parent`: The Port, Interface or Module object this Generic belongs to.

The value check function for Generics can only check constant values assigned in the user script. This function will normally validate any value. It can be exchanged by the user. We expect users to add custom value check functions in the module specification files, where useful. For example: If a Generic specifying the data width of a bus can only have values divisible by eight, the module developer may associate a value check function that tests for this to the Generic. If a user then tries to assign an invalid value to the Generic, an error message will automatically be generated. The value check function can be set using the Generic object's method:
`generic.set_value_check(function)`
Now the value check function will be triggered, if the Generic's value is changed using the `generic.set_value(value)` method or the `module.set_generic_value(...)` method.

6.2.4.6. SlaveRegisterInterface

The SlaveRegisterInterface class is an abstraction of the Interface class and generally very similar. It contains a few extra methods for dealing with register specific tasks.
Here is a list of the ports that an automatically managed slave register interface requires:

- `slv_status_reg`: Data transport to software. An array of 32 bit registers.
- `slv_ctrl_reg`: Data transport to hardware. An array of 32 bit registers.
- `slv_reg_modify`: Data modify enable signals for the status registers. A std_logic_vector as wide as the number of registers.
- `slv_reg_config`: A port to export the register configuration. An array of two bit std_logic_vectors, designating how the registers are configured.

The register interface also requires a constant to be set.
Important: The constant `slave_register_configuration` must be set in the architecture description of the modules VHDL toplevel, before any `begin` keyword! Otherwise Automatics won't find it, which results in the register interface not being recognized.
The architecture of the register managers employed by Automatics requires, that all modules with register interfaces are potentially assigned the same number of registers. This is increased by powers of two. This means that each module gets assigned an address space of the same size - as large as the module with the most registers. If you have a module that requires a large amount of registers, you might want to consider using multiple register interfaces, if you run into the problem that the address space is getting too small. Note that multiple register interfaces should not significantly increase the amount of hardware generated.

Important: To assign multiple register interfaces to a module, duplicate all ports and the configuration constant, then differentiate them *by suffix only*! Otherwise certain generated signals will have duplicate names, causing the synthesis to fail. This is a quirk of Automatics and might be fixed in a future version.

For more information about the register interface and how to integrate one into your module, read section 5.1.6.

6.2.4.7. AsModule

The class AsModule is the main class used to represent any *ASTERICS* module internally in Automatics. Using the method `chain.add_module()` in the user script, duplicates the AsModule object of the desired module, adds it to the processing chain object and returns a reference to the duplicate. This allows the user to directly access methods from AsModule.

Important attributes of Automatics class AsModule:

- `name`: The custom name given to the module by the user or automatically by Automatics.
- `entity_name`: The VHDL entity name of the toplevel VHDL file for this module. Automatics uses this name to search for modules when the `add_module()` method is used.
- `module_dir`: Where the source files for this module are stored on your system.
- `files`: A list of paths for each source file that is part of this module.
- `ports`: A list of ports of this module that are not part of any interface and are also not recognized as standard ports.
- `standard_ports`: A list of ports of this module that are recognized by Automatics as standard ports, such as `clk` or `reset`. This means that Automatics can automatically connect them in a known manner.
- `interfaces`: A list of interfaces that are part of this module.
- `register_ifs`: A list of slave register interfaces that are part of this module.
- `generics`: A list of Generics that are part of this module.
- `interface_templates`: A list of interface templates that are part of this module specifically. A second list of interface templates exists, which is accessible class-wide using `AsModule.interface_tempaltes_cls`.
- `dependencies`: A list of module entity names which this module is dependend upon.

6.2.4.8. AsModuleLibrary

The AsModuleLibrary is a class that contains a list of module repository objects (AsModuleRepo), each containing a list of AsModule objects. The module repositories are filled with module objects when Automatics scans your *ASTERICS* installation for modules. This happens when the `chain` object is created using `asterics.new_chain()` or when adding a new repository using `asterics.add_module_repository()`. The folder structure is scanned for modules and their module specification scripts (see section 6.2.5). Each specification is parsed into a new AsModule object based on just the specification script and the module's VHDL toplevel file. Each location on your system that Automatics scans is representated by a new AsModuleRepo object (unless you specify the same repository name).

The AsModuleLibrary is only instantiated once per run of Automatics and can be queried for specific modules or list modules from specific repositories or all available modules. The library does not usually need to be used directly by users, instead it is called internally via calls to the processing chain object or in the interactive modes.

6.2.4.9. AsProcessingChain

This class represents the entire image processing chain. The AsProcessingChain contains most methods that are responsible for connecting and managing the contained modules, their interfaces, ports and generics.

Here is a list of the most important attributes:

- `library`: A reference to the AsModuleLibrary, storing all available *ASTERICS* modules
- `top`: A reference to the *ASTERICS* VHDL toplevel module group.
- `as_main`: A reference to the as_main module group where all *ASTERICS* processing modules are instantiated and connected.
- `modules`: A list of all regular *ASTERICS* modules part of this processing chain.

The processing chain object provides an extensive interface for describing and building an ASTERICS IP-Core (see section 6.2.6).

6.2.5. Using New Modules in Automatics

This section describes the process of integrating a new *ASTERICS* hardware module, such as filter module or similar, into the generation procedure of Automatics.

For this to work, certain prerequisites must be fulfilled:

- The hardware module must be written in a compatible VHDL formatting style. Use existing modules to determine if this is true for your module. Also refer to section 6.2.5.1 for a list of special rules that must be followed for Automatics to correctly import your module.
- For the generator to recognize interfaces, port names of interfaces must have common pre- and/or suffixes. The pre- and/or suffixes `"in"` and `"out"` are always ignored.

- The module must be stored in a folder structure mimicking that of the existing modules (described below).

To include a custom module, follow this brief guide:
In the user script, use the method `add_module_repository("path", "name")` of the `asterics` Python module, to point Automatics to a ”modules” folder (the name doesn't matter) which contains one or more folders, each for one or more modules. Every subfolder of the ”modules” folder contains a folder named `"hardware"`, which in turn contains a folder named `"automatics"`. In this `"automatics"` subfolder, add a short Python script for each module you want to define in this subfolder of the `"modules"` folder. For a visual overview, refer to Figure 6.1. You should copy the script template `as_automatics_module_spec_template.py`. Rename the template according to this naming scheme (important!):
`"as_[module name]_spec.py"`
Fill out the module specification template, substituting markers such as `"<module name>"` or `"<module author>"` and follow the instructions in the comments of the Python code. You should be editing the function `get_module_instance()`. Automatics will search for files in the `automatics` folder that start with `as_` and end in `_spec.py`, running the `get_module_instance()` function to import the AsModule object. Make sure that you don't delete the command `return module` at the very end of the function!

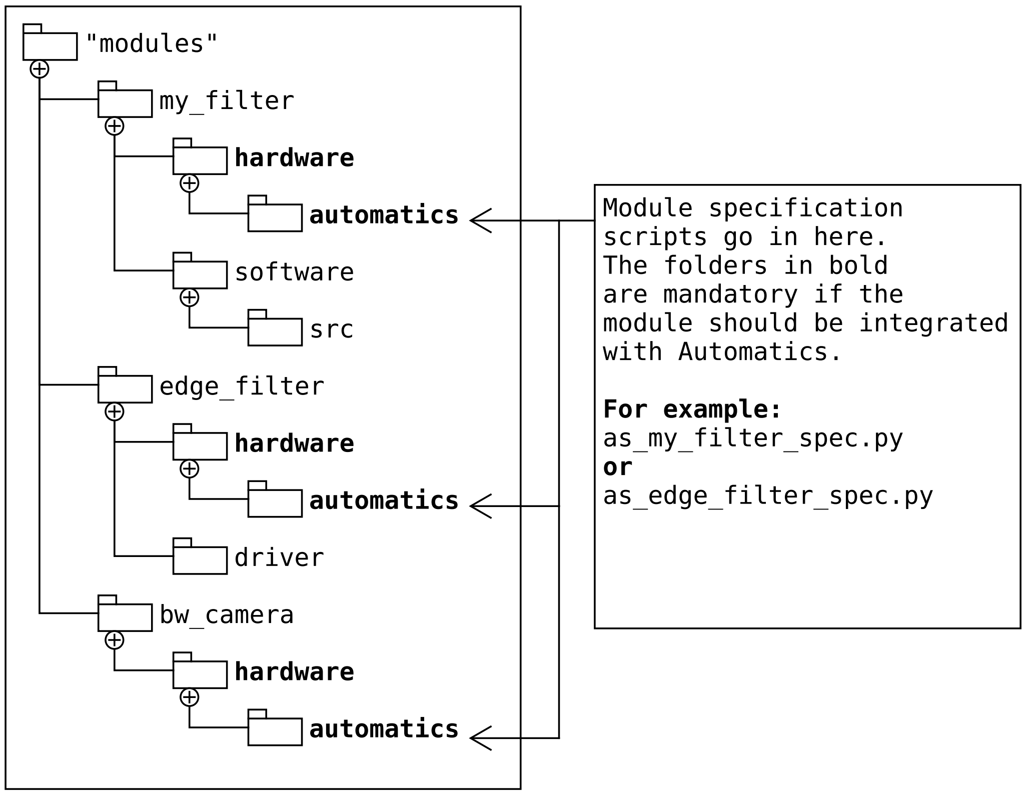

Figure 6.1.: Folder structure to follow when developing a new module that should be integrated into Automatics.

As mentioned in section 6.2.4.4, in the module specification files, before the call to `module.discover_module`, you can add Interface templates to just this module by using the `module.add_local_interface_template()` method.

Most other configuration options, such as modifying Ports, Generic and Interface settings, must be done after the call to `module.discover_module()`, as this is the method Automatics uses to create all the Port, Generic and Interface objects. See the specification template file and the Doxygen documentation or the Python docstrings for explanations of the functionality of the available methods of the different Automatics objects. You will want to look at the classes Port, Generic, Interface, AsModule and AsRegisterInterface (and maybe Constant).

To quickly get started with a custom module, you may use the modules included with *ASTERICS* , contained in `<asterics>/modules` as a starting point.

6.2.5.1. VHDL Coding Rules for *ASTERICS* Modules for Use with Automatics

This section lists a few rules that must be followed in VHDL code for Automatics to correctly import a processing module. These result from quirks of the VHDL Reader used in Automatics to parse the VHDL code.

- To end a VHDL `entity` you must use either the entity name or the keyword `entity` after the `end` keyword. Eg.: For `entity example is [...]` use either `end example;`, `end entity;` or `end entity example;`.
- `port` and `generic` definitions in an `entity` must be in a single line.
- The beginning and end of both `entity` and `architecture` must each be in a single line.
- Port and generic names must not contain any of the following keywords: "entity", "architecture" and "begin".
- Port and entity names should be kept all lower-case (within Automatics, they are handled and should be imported as all lower-case).
- Generic names should be kept all upper-case (within Automatics, they are handled and should be imported as all upper-case).
- The constant for the slave register configuration must be defined and contain the assignment operator (`:=`) in a single line. Further value assignments of the array may be spread out onto multiple lines.
- Futher, the constant must be placed before *any* `begin` keyword, including `begin` as it appears in component declarations.

Note: Automatics only consideres the first entity and architecture in each VHDL file as an *ASTERICS* module.

All VHDL code after the `begin` statement in the `architecture` is ignored by Automatics. No special considerations have to be followed for that code.

6.2.6. List of Configuration Methods

This section lists configuration methods intended for use in the user script, to configure and build an *ASTERICS* processing chain. The methods are listed in table 6.3.

Object	Method	Parameters	Description
`AsProcessing Chain`	`add_module()`	`"entity name" , "user name"`	Add the module `"entity name"` from the module library to the processing chain. Use the reference name `"user name"` (optional, if ommitted, Automatics will choose a name for you).
`AsProcessing Chain`	`set_asterics_ base_address()`	`"base address" , "address space size"`	Redefine the *ASTERICS* base address for communication between the *ASTERICS* modules and software. Set the available address space. *Important:* Note that the base address must begin with all bits in the available space as zero! Eg.: OK: Base = `0x431C0000` and Size = `0xFFFF`, not OK: Base = `0x431C8000` and Size = `0x8FFF` (must be Base = `0x431C0000`)
`AsProcessing Chain`	`connect()`	`"from", "to"`	Connect module, interface or port `"from"` to module, interface or port `"to"`. The direction of data flow is assumed to be from `"from"` to `"to"`.
`AsProcessing Chain`	`write_hw()`	`"folder", "use_symlinks", "force"`	Generate only the hardware source output files to *folder*. By default existing module source files are linked to the output folder. To copy them instead, use *use_symlinks=False*. To allow Automatics to clean the output folder by permanently deleting contained files, set *force=True*.

Object	Method	Parameters	Description
`AsProcessing Chain`	`write_sw()`	`"folder", "use_symlinks", "force", "driver_module_ _dirs"`	Generate only the software source output files to *folder*. By default existing module source files are linked to the output folder. To copy them instead, use *use_symlinks=False*. To allow Automatics to clean the output folder by permanently deleting contained files, set *force=True*. If you want source files sorted into subfolders per module, set *driver_module_dirs=True*.
`AsProcessing Chain`	`write_asterics_ _core()`	`"folder", "use_symlinks", "force", "driver_module_ _dirs"`	Runs both `write_hw()` and `write_sw()`. The parameters are identical in functionality to those methods.
`AsProcessing Chain`	`write_ip_core_ _xilinx()`	`"folder", "use_symlinks", "force", "driver_module_ _dirs"`	Runs both `write_hw()` and `write_sw()`. The parameters are identical in functionality to those methods. In addition, the resulting IP-Core is packaged using Vivado. Vivado has to be installed on your system and sourced in the console environment you used to run Automatics for this step to complete.

Object	Method	Parameters	Description
AsProcessing Chain	write_system()	"folder", "use_symlinks", "force", "driver_module_dirs", "add_vears"	Runs both `write_hw()` and `write_sw()`. The parameters are identical in functionality to those methods. The outputs are generated into a system template prepared for a Vivado-style FPGA project. In addition, the resulting IP-Core is packaged using Vivado. Vivado has to be installed on your system and sourced in the console environment you used to run Automatics for this step to complete. Furthermore, the VEARS IP-Core for video output is added to the project. To omit it, set *add_vears=False*.
AsProcessing Chain	write_system_graph()	"out_file", "show_toplevels", "show_auto_inst", "show_ports"	Generate and write a graph representation of the `system`, a "chain" object, passed to the function to the file `out_file`. The other parameters are all `False` by default and may be turned on to make the graph more detailed. `show_toplevels` adds the "meta modules" used to connect the processing modules to the graph. `show_auto_inst` adds the automatically instantiated modules, which are typically not of interest to the user, as they usually perform data management tasks such as converting between interfaces. `show_ports` adds a list of all ports to each edge of the graph which represent interfaces.
AsModule	connect()	"object"	Shorthand for `AsProcessingChain.connect(self, object)`.

Object	Method	Parameters	Description
`AsModule`	`get_port()`	`"port name"`	Return the Port of the module with the `code_name` or `name` attribute `"port name"`.
`AsModule`	`get_interface()`	`"interface name"`, `"direction"`, `"interface type"`	Return the Interface of the module with the `name` and `pre-/suffix` attributes matching `"interface name"`. Optionally exclude any interfaces from the search not matching with `"direction"` and/or `"interface type"`.
`AsModule`	`get()`	`"name"`, `"direction"`	Shorthand for a combination of `get_interface` and `get_port`. Returns the first interface matching `"name"` and/or `"direction"` or, if none was found, searches ports instead.
`AsModule`	`get_generic()`	`"generic name"`	Return the Generic of the module with the `code_name` or `name` attribute `"generic name"`.
`AsModule`	`make_port _external()`	`"port name"`, `"value"`	Search for a Port matching `"port name"` and change it's `ruleset` to have Automatics make it external (default), meaning it will be available on the interface of the resulting IP-Core. Alternatively, make an external port internal by setting `"value"` to `False`.

Object	Method	Parameters	Description
`AsModule`	`make_interface _external()`	`"interface name"`, `"direction"`, `"if_type"` `"value"`	Search for an Interface matching `"interface name"` (and optionally `"direction"` and `"if_type"`) to have Automatics make it external (default), meaning it will be available on the interface of the resulting IP-Core. Alternatively, make an external interface internal by setting `value` to `False`. Note that you will have to either provide the parameter name `(value=False)` or use all parameters.
`AsModule`	`set_port _fixed_value()`	`"port name"`, `"value"`	Search for a Port matching `"port name"` and change it's `ruleset` to have Automatics set it to the fixed value `"value"`. Automatics will directly insert the value of `"value"` into VHDL code, so make sure it won't break VHDL syntax.
`AsModule`	`set_generic _value()`	`"generic name"`, `"value"`	Search for a Generic of the module with the `code_name` or `name` attribute `"generic name"` and set it's `value` attribute to `"value"`. Note that `value` will be directly inserted into VHDL code later, so make sure the provided value won't break VHDL syntax.
`AsModule`	`port_rule _add()`	`"port name"`, `"condition"`, `"action"`, `"priority"`	Search for a Port matching `"port name"` and add to it's `ruleset` the new rule `"condition" -> "action"`. This rule will have top priority, so will be executed first, unless you set `"priority"` to `False`.
`AsModule`	`port_rule _remove()`	`"port name"`, `"condition"`, `"action"`	Search for a Port matching `"port name"` and remove the rule `"condition" -> "action"` from it's `ruleset`, if it exists.

Object	Method	Parameters	Description
`AsModule`	`port_rule _overwrite()`	`"port name", "condition", "action"`	Search for a Port matching `"port name"` and overwrite all `actions` for `"condition"` in it's `ruleset`. Then add the rule `"condition" -> "action"`.
`AsModule`	`make_generic _external()`	`"generic name"`	For the Generic matching `"generic name"`, set its attributes so that it will be propagated to toplevel, allowing your synthesis tool to set the value using the *ASTERICS* IP-Core.
`Interface`	`make_external()`	”value”	Set the necessary attribute to make this Interface available externally. Automatics will connect it to the VHDL toplevel, so it is in the interface of the resulting IP-Core. Pass `False` to make external interfaces internal.
`Interface`	`instantiate _module()`	`"entity name", "group name"`	Set the necessary attributes to have Automatics automatically add and instantiate the *ASTERICS* module `"entity name"` and connect the interface to it. The `group name` defines where in the hardware design the module is instantiated. This particular functionality is currently not implemented completely - the only two options for the `group name` are: `"asterics"`, to instantiate to the toplevel (default) and `"as_main"` to instantiate the module where the *ASTERICS* modules are connected. This functionality is useful, for example, to instantiate necessary bus manager modules or something similar.

Object	Method	Parameters	Description
`Interface`	`instantiate_no_module()`	None	Remove a configuration for an automatic instantiation of a mode from this interface object.
`Generic`	`link_to_generic()`	`"link generic"`	Set the necessary attributes to have Automatics link the value of this Generic to the Generic `"link generic"`, using the `code_name` attribute, of higher modules. This can be useful, for example, to set the data bus width of multiple modules, using a user added Generic on the toplevel. (Note: This feature has not been tested extensively.)
`Port`	`set_port_type()`	`"port type"`	Manually define the Port's `port_type` attribute. Useful if a Port is determined to have the wrong port type by Automatics. Valid values: `single`, `external`, `interface`, `register`, `signal`, `glue_signal`.
`Port`	`remove_condition()`	`"condition"`	Remove all rules with the condition `"condition"` from this Port's `ruleset`. You may use `module.port_rule_overwrite(<port name>, <condition>, "none")` for the same effect.
`Port`	`update_ruleset()`	`"list"`	Use this method to quickly add multiple rules to this Port's `ruleset`. Note that any invalid rules are quietly skipped. Any iterable list is accepted. Note that the rules are expected to use the `Port.Rule namedtuple` (check the file `as_automatics_port.py`)!

Object	Method	Parameters	Description
`Port`	`set_ruleset()`	`"list"`	Use this method to quickly replace this Port's `ruleset`. Note that any invalid rules are quietly skipped. Any iterable list is accepted. Note that the rules are expected to use the `Port.Rule namedtuple` (check the file `as_automatics_port.py`)!
`Port`	`make _external()`	None	Modify this Port's `ruleset` to have Automatics make it external. Same as AsModule's `make_port_external()` method.

Table 6.3.: List of configuration methods intended for use in the user script.

6.2.7. List of Useful Methods of Automatics Objects

This section lists a selection of methods applicable to all important Automatics objects. These methods are intended to give the user all the necessary information to build an *ASTERICS* processing chain, without having to read VHDL code. These functions are typically not required to be used to build a processing chain using Automatics, but are rather additional tools to achieve more complex designs, explore the generated chains and gain additional insight into the generated code. The methods are listed in table 6.4.

Object	Method	Parameters	Description
`AsAutomatics`	`add_module _repository()`	`"path"`, `"name"`	Have Automatics scan a folder for modules and add them to the repository `"name"`. This functionality should normally be invoked via the `asterics.py` Python module. The Automatics object instantiated by `asterics.py` can be accessed by `asterics.Auto`.

Object	Method	Parameters	Description
`AsAutomatics`	`new_proc_chain()`	`"function"`	Set up a new processing chain, using the definition function `"function"`. The new `AsProcessingChain` object will be assigned to attribute `current_chain` of the AsAutomatics object used to call the method. Returns `True` on success. This functionality should normally be invoked via the `asterics.py` Python module.
`AsModule Library`	`list_modules()`	`"verbosity"`, `"repo"`	List all modules of all repositories. Change the detail level using `"verbosity"`, valid range [0, .., 3]. Use `"repo"` to only list from specific repositories. Standard *ASTERICS* modules are stored in repository ”default”.
`AsModule Library`	`has_module()`	`"name"`, `"repo"`	Returns `True` or `False` depending on if module `"name"` is present in the module library. Limit the search to specific repositories using the `"repo"` parameter.
`AsModule Library`	`get_module_template()`	`"name"`, `"repo"`	Returns a reference to the module object matching `"name"` in the module library. Limit the search to specific repositories using the parameter `"repo"`. The returned object should not be modified or added to a processing chain, instead use `get_module_instance()`!

Object	Method	Parameters	Description
`AsModule Library`	`get_module _instance()`	`"name"`, `"repo"`	Returns a copy of the module object matching `"name"` in the module library. Limit the search to specific repositories using the parameter `"repo"`. The returned object is a copy of the module object in the library and can safely be modified and used in processing chains. If you only need read access, use `get_module_template()` instead, for better performance.
`AsProcessing Chain`	`list_address _space()`	None	Call after building the processing chain to list the addresses and types of all allocated slave registers of the included modules.
`AsProcessing Chain`	`get_module()`	`"name"`	Return the module object for the module `"name"` from the list of modules currently in the processing chain.
`AsProcessing Chain`	`auto _instantiate()`	None	Instantiate all modules defined through the `instantiate_in_top` attribute of the respecitve interfaces of the modules for this processing chain. Returns a list of the modules instantiated. Should not normally be called manually. This method is useful if you need to modify the parameters of an auto-instantiated module, but do not want to manually instantiate it. Call this method right before calling a `chain.write_...` method.
`AsModule`	`list_module()`	`"verbosity"`	List information extracted from the VHDL toplevel file of this module. Use `"verbosity"`, set between zero and two, to list with increasing detail.

Object	Method	Parameters	Description
`AsModule`	`get _unconnected _ports()`	None	Return a list of unconnected ports of this module. Caution! This method is known to return many false positives!

Table 6.4.: List of useful methods intended for use by the user while building a processing chain.

6.2.8. Interactive Mode

As the full interactive mode is currently still in development, many of the planned features are not yet implemented. However, at this time, functions for exploring the internal representations of any modules, that Automatics has scanned, listing functions and a function for scanning other folders are available.
After sourcing *ASTERICS* (`source <asterics>/settings.sh`), the interactive mode is available by calling `as-module-browser-cli` in the console. Once loaded, the available functions can be listed with the command `as_help()`. Details about the functions can be shown using `<function name>?`.

Prerequisites:

- An *ASTERICS* installation
- Python 3.5 or higher

6.2.8.1. Graphical Mode

For the interactive mode a GUI is also in development. At the moment it still contains bugs and is far from feature complete. It does however, provide a more convenient method for exploring *ASTERICS* modules.

Prerequisites:

- An *ASTERICS* installation
- Python 3.5 or higher
- The PyQt5 GUI library. Installation using `pip install pyqt5`.

You can start the GUI after sourcing the *ASTERICS* settings file by running `as-module-browser` in your console. We currently do not provide a user guide for the GUI mode as it is very much subject to change and does not provide too extensive functionality.

6.3. 2D Pipeline Generator

by Alexander Zöllner

6.3.1. Brief Description

The *2D Pipeline Generator* is a script-based tool for building a 2D pipeline in a semi-automatic way. Required filter masks for the 2D window modules are extracted from images using the program Gimp. The resulting 2D pipeline consists of storage elements for each data type and a number of *2D Window Interfaces*.

6.3.1.1. Hardware Components

Table 6.5 lists the required hardware components for the 2D Pipeline Generator in order to build the pipeline.

Component	Purpose
`myLittleHelpersPkg.vhd`	Defines several mathematic functions such as finding the exponent of two required for representing a given integer.
`dram_shift_reg.vhd`	Implements the logic required to store data in the 2D-sliding window buffer using distributed RAM (DRAM), i.e. slice registers. DRAM is used for small sections up to 100 elements.
`bram_shift_reg.vhd`	Implements the logic required to store data in the 2D-sliding window buffer using Block-RAM (BRAM). BRAM is used for long sections which require more than 100 elements.
`myBRAM.vhd`	Implements the actual storage element in BRAM.

Table 6.5.: Hardware components required for the 2D Pipeline Generator

6.3.1.2. Generator Input

In order to build the 2D pipeline, the 2D Pipeline Generator has to determine the required interfaces. Therefore, an image for each filter mask has to be provided using the image program Gimp (of version 2.8). In the first step, a new image has to be created. The height of the image has to match the one of the filter mask and the width the resolution of the image in x direction. For the filter mask, a new layer has to be created, matching the size of the image. The layer should be named appropriately, since the generator uses the same name for the corresponding interface of the filter mask. Further, the background layer has to be deleted. The in- and output positions of the 2D window module are chosen by coloring the associated pixel, as shown in table 6.6. Reference pixels are solely for visual user assistance, e.g. marking the center pixel of a Gaussian filter mask. In a final step, the generated image has to be exported in a png format.

	In	Ref.	Out
In	0xff0000	0xffff00	0xff00ff
Ref.	0xffff00	0x00ff00	0x00ffff
Out	0xff00ff	0x00ffff	0x0000ff

Table 6.6.: Color encoding used for I/O configuration

The in- and output ports of the filter masks have to be associated with their corresponding data type to infer the required amount of storage elements and connecting them correctly. For this reason, an input file, namely *data.txt* has to be provided, which contains labels for all data types with their associated bit width. An entry has the format `<label>:<bit-width>`, where the *label* must only contain letters (a-z, A-Z) and the *bit-width* only digits (0-9).

Data types are associated with the filter masks by assigning the labels to the ports of a given filter port. This is done in the *connections.txt*, which is an output of the 2D Pipeline Generator.

6.3.1.3. Scripts

Name	Purpose
`generateConnectionFile.py`	Generates the file connections.txt from the png files containing the filter masks and data.txt for the data types.
`generateDataPipeline.py`	Generates the actual 2D pipeline.
`dataPipeline.py`	Implements the functionality of the 2D Pipeline Generator and provides methods for generateConnectionFile.py and generateDataPipeline.py

Table 6.7.: Scripts of the 2D Pipeline Generator

6.3.2. Output Products

Name	Purpose
`connections.txt`	Lists ports of all filter masks. Has to be edited by user.
`dataLayers.log`	Lists the BRAM resource consumption as well as implementation details for individual layers.

Name	Purpose
`dataPipeline.vhd`	The vhdl module containing the 2D pipeline to be instantiated by the user.
`dataPipeline_mapping.vhd`	Provides signal declarations and component instantiation in vhdl, which can be copied in the user design.

Table 6.8.: Output products of the 2D Pipeline Generator

7. Basic Modules

As already noted, there is a bunch of different modules for image processing in the framework. The functionality and usage of these modules will be described in the following sections.

7.1. as_global_processing

by Alexander Zöllner

7.1.1. Brief Description

The `as_global_processing` is the software driver for the `asterics_state/control` register for requesting the state of the *ASTERICS* -chain as well as for performing a reset on it. Access functions are provided by this software driver for accessing the aforementioned register correctly without having to remember the actual bit fields of the hardware mapping. It is expected that the status and control fields of the hardware modules within the *ASTERICS* -chain are mapped correctly to the fields of the `asterics_state/control` register.

7.1.2. Register Interface

The register space is part of the *Common Control and Status Registers* described in Chapter 5.1.1.

7.2. Input/Output Modules

7.2.1. as_memreader

by Alexander Zoellner

7.2.1.1. Brief Description

The `as_memreader` is a hardware module for efficiently transferring data from main memory to the FPGA. The module utilizes a bus master interface to memory and an *as_stream* interface to programmable logic. It transfers chunks of data , so called *sections*, by using burst accesses whenever possible. *Sections* are defined by a start address and size. Multiple *sections* are also supported, which may include a constant offset (i.e. the difference between the start address of two consecutive *sections*) in between, allowing to read rectangular sub-images or skipping certain parts of data layouts. In order to decrease the amount of time spent idle, the following *section* can be programmed during an ongoing operation by overwriting the hardware registers of the `as_memreader` and setting its `go` flag again. The module proceeds with the next data transfer after the current section has been completed, i.e. all data has been read. The module provides a register for tracking the progress of the current *section*, which holds the next address the module is going to read from.

7.2.1.2. Architecture

Figure 7.1 shows the architecture of the `as_memreader` module in simplified form to emphasize the main parts of the module, which are required for performing data transfers. The `as_memreader` uses a *Master Interface* for obtaining data from memory. This interface comprises only signals which are required for configuring the access to memory, such as the number of bytes to be transferred or when to start a data transfer. The actual memory bus access is performed by a *Bus Translation* module, which is connected to the memory bus interface and the `as_memreader`. Data is intermediately stored in a *FIFO Buffer* first, before passing it to the subsequent hardware module. As aforementioned, the memory modules usually transfer a chunk of data at once, utilizing burst accesses. However, the subsequent hardware module may not be able to handle the amount of data at the same pace as the memory module. The *FIFO Buffer* is used to match the processing speed of the subsequent module, by being able to temporarily suspending data transfers towards the module. The `as_memreader` continuous to request data from memory, as long as the fill level of the *FIFO Buffer* has not reached a certain threshold. Hardware modules are connected to the `as_stream` (see Chapter 5.3) interface of the `as_memreader`. For configuring the `as_memreader`, a *Register Interface* is utilized, which provides a number of hardware registers. These registers can be accessed by hardware and software alike, for exchanging control and status information. During an ongoing operation, certain information must not be changed for the `as_memreader` module to operate correctly. For this reason, information regarding the specifics of the operation, such as number of *sections* or its size, have to be kept stable. This is accomplished by copying critical information to *Shadow Registers* at the start of an operation. As a side effect, this allows the software to replace the associated hardware registers during an ongoing operation, which is used for queuing the subsequent operation for transferring data. The *Memory State Machine* controls the data flow and operation within the `as_memreader` module. This part is responsible for determining the point at which the contents of the *Shadow Registers* are replaced, publishing status information to software via the Register Interface as well as accepting control information from it. For the actual data transfer, the *Memory State Machine* is responsible for setting the appropriate signals at its *Master Interface* and `as_stream` interface, depending on the fill level of the *FIFO Buffer*. Additionally, it evaluates the `as_stream` signal for the `as_memreader`. The *Address Generator*, as the name suggests, calculates the address required for the individual memory accesses as well as the number of bytes to be transferred. The required information are obtained from the *Shadow Registers*. Towards software, the address used for the next data transfer is published to software via the *Register Interface*.

7.2.1.3. Pre-Synthesis Options

Name	Range	Description
REGISTER_BIT_WIDTH	Positive integer value	Bit width for the slave registers used by this module (usually 32 bit).
DOUT_WIDTH	Positive integer value	Bit width of the data port of this module. Currently uses the same bit width as the memory bus interface (usually 32 or 64 bit).
MEMORY_DATA_WIDTH	Positive integer value	Bit width of the data port to memory. Depends on the setting of the memory bus (usually 32 or 64 bit).
MEM_ADDRESS_BIT_WIDTH	Positive integer value	Bit width of the address port of the memory bus. Depends on the memory bus interface (usually 32 bit).
BURST_LENGTH_BIT_WIDTH	Positive integer value	Bit width of the memory bus to configure the burst size (currently 12 bits).
MAX_PLATFORM_BURST_LENGTH	Positive integer value	Highest possible burst length support by the memory bus in bytes (currently: 256).
FIFO_NUMBER_OF_BURSTS	Positive integer value	Minimum size of the internal data fifo. Actual size is to the power of 2 required for fitting the chosen size.
SUPPORT_MULTIPLE_SECTIONS	Boolean	Enables regular address jumps for reading data from memory.
SUPPORT_VARIABLE_BURST_LENGTH	Boolean	Allows to configure actual burst length at runtime.

SUPPORT_INTERRUPTS	Boolean	Allows the module to generate interrupt events.
SUPPORT_DONE_IRQ_SOURCE	Boolean	Module generates an interrupt event at the end of a *section*. SUPPORT_INTERRUPTS has to be enabled.

7.2.1.4. Register Space

Name	Relative Address	Width	Description
`state/control`	0x0	32	Status information of the `as_memreader` and controlling the operation of the module. The lower half of the register is used for status information, the upper half for configuration.
`section addr`	0x4	32	Memory address at which the module starts to read data from.
`section offset`	0x8	32	Address offset in byte for regular address jumps. The offset is the distance between the start addresses of two consecutive *sections*.
`section size`	0xC	32	Size of a *section* in byte to be read from memory. The size has to be a multiple of the configured data bus width in byte.
`section count`	0x10	32	Number of *sections*. If `SUPPORT_MULTIPLE_SECTIONS` is not set, this register is ignored and a single *section* is assumed. Otherwise, an address jump is performed between two consecutive *sections*.
`max burst length`	0x14	32	Number of bytes to be requested for a single memory bus access. Has to be a multiple of `MEMORY_DATA_WIDTH` and must not exceed `MAX_PLATFORM_BURST_LENGTH`.

`current hw addr`	0x18	32	Next memory address the module is going to read from. If the `as_memreader` is not programmed this address is 0x0. For multiple *sections*, it points to the last *section* start address + offset after the last *section* has been served.

Table 7.2.: Register overview of the `as_memreader` module.

Field Name	**Bit Index**	**Type**	**Reset Value**	**Description**
`go`	17	wo	0x0	If set, the `as_memreader` starts transferring data to memory.
`reset`	16	wo	0x0	Resets the module and clears all remaining data in the *FIFO Buffer*.
`pending go`	5	ro	0x0	If set, the next data transfer has already been programmed.
`busy`	1	ro	0x0	The `as_memreader` is currently operating.
`ready`	0	ro	0x0	The `as_memreader` is currently idle.

Table 7.3.: Bit field overview of the combined `state/control` register of the `as_memreader`.

Field Name	**Bit Index**	**Type**	**Reset Value**	**Description**
`section address`	31:0	wo	0x0	Start address for reading data from memory.

Table 7.4.: Bit field overview of the `section addr` register of the `as_memreader`.

Field Name	**Bit Index**	**Type**	**Reset Value**	**Description**

`section offet`	31:0	wo	0x0	The offset is the distance between two start addresses in byte.

Table 7.5.: Bit field overview of the `section offset` register of the `as_memreader`.

Field Name	Bit Index	Type	Reset Value	Description
`section size`	31:0	wo	0x0	Size of a *section* in byte to be read from memory.

Table 7.6.: Bit field overview of the `section size` register of the `as_memreader`.

Field Name	Bit Index	Type	Reset Value	Description
`section count`	31:0	wo	0x0	Number of *sections.*

Table 7.7.: Bit field overview of the `section count` register of the `as_memreader`.

Field Name	Bit Index	Type	Reset Value	Description
`max burst length`	31:0	wo	0x0	Number of bytes to be requested for a single memory bus access.

Table 7.8.: Bit field overview of the `max burst length` register of the `as_memreader`.

Field Name	Bit Index	Type	Reset Value	Description
`current hw addr`	31:0	ro	0x0	Next address the `as_memreader` is going to read from.

Table 7.9.: Bit field overview of the `current hw addr` register of the `as_memreader`.

7.2.1.5. Behavior

The bit fields of the `control` and `state` registers are also represented internally within the module by signals with the same name. When referring to the bit field, the name is written in lower case and in upper case for the signal. Flags refer to the bit fields.

When the module receives a reset request, either by using the hardware port or the `reset` flag of the *State Control* register, ongoing bus accesses are terminated and all internally stored configuration settings are cleared after a single clock cycle. Additionally, the module requests to clear the `go` and `reset` field. The configuration registers towards software are left as is. Simultaneously, the data in the *FIFO Buffer* is discarded and

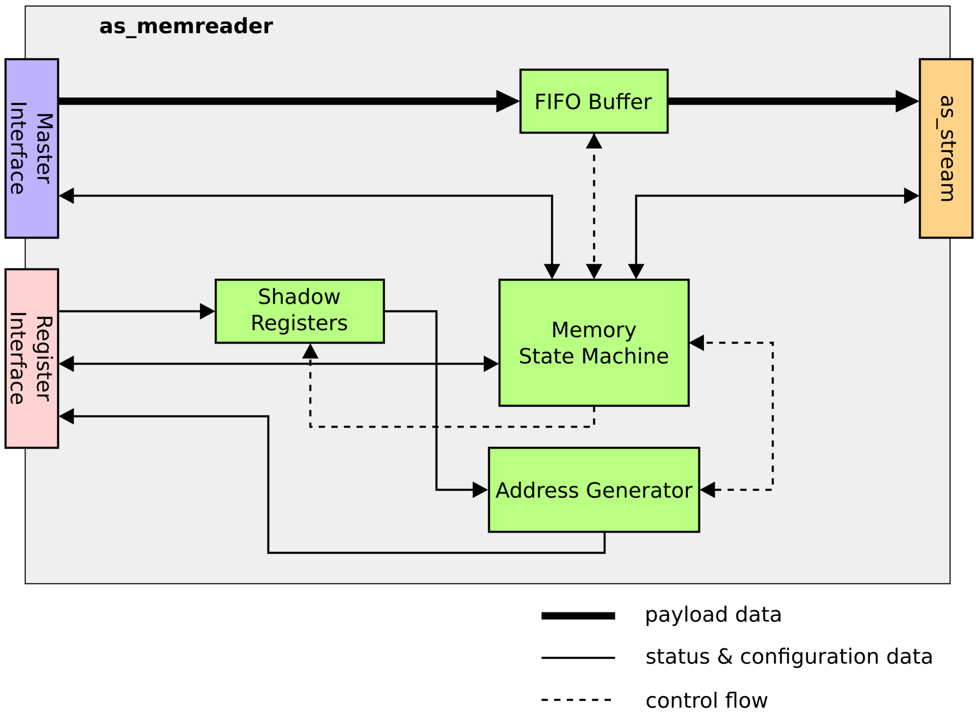

Figure 7.1.: Architecture of the `as_memreader` module

the address generator is reset. The reset persists only for one clock cycle if it has been requested by software, otherwise up to the point it is lifted by hardware. During the reset, neither of the modules' status flags is set. After the reset has been lifted, the `as_memreader` assumes its idle state.

In the modules' idle state, the `ready` flag is set and the configuration registers are continuously copied into internal shadow registers. Upon setting the `go` field, the current configuration is used for setting up data transfers after one clock cycle. Further, the `ready` flag is unset and the `busy` flag is set instead, which is kept throughout the whole operation. The `as_memreader` requests the slave logic to clear the `go` flag and starts accessing the master memory bus interface. During operation, the configuration registers can be overwritten and `go` flag set again, to seamlessly queue the next data transfer. If the `go` flag was set in this manner, the module sets its `PENDING GO` signal. Since the `as_memreader` copies the configuration only at the start, it would be possible to change the configuration for the following transfer after having already set the `go` flag again. However, it is strongly recommended to configure the data transfer before queuing the next transmission. As soon as the `as_memreader` has read all requested data for the current operation, it sets the `ready` flag again and returns to its idle state. Here, the `busy` and `ready` flag are set simultaneously for a single clock cycle, which is exclusively used if another hardware module interfaces the `as_memwriter` instead of software. This is mainly for backwards compatibility and may be subject to change in the future. If the next `GO` has already been set when entering the idle state, the module proceeds in the aforementioned manner without delay. The `PENDING GO` (since `GO` is unset) is unset again.

The `as_memreader` is a *stall-absorbing* module (see Chapter 5.3.4), which means it accepts requests to suspend data transfers by the following module. After receiving a stall request, the `as_memreader` stops transferring data at its *as_stream* output interface on the following clock cycle, until the stall is lifted again. Since the module utilizes a

FIFO Buffer, it continues to read data from memory if there is remaining space. The fifo is considered full, if it can only hold the equivalent of one additional maximal possible burst length amount of data. Since the actually utilized burst length may vary (if `SUPPORT_VARIABLE_BURST_LENGTH` is set), this method allows to prevent *FIFO Buffer* overflow without having to check the precise amount of space left for each memory bus request. This method has mainly been chosen to prevent the module from inefficiently reading multiple small amounts of data in order to utilize the whole *FIFO Buffer*. Instead, if the aforementioned threshold has been reached, the `as_memreader` suspends reading data from memory until the *FIFO Buffer* has output some data. As a side effect, some FPGA area has been saved which would be required for the comparison.

The `as_memreader` module organizes its data transfers in one or more *sections*, where each one is a physically continuous chunk of data in memory. If more than one *section* is required, the configuration option `SUPPORT_MULTIPLE_SECTIONS` has to be set at synthesis time. The register *Section Address* defines the memory address of the first *section*, whereas *Section Offset* is used for the following ones. The latter is the difference of the start addresses of two consecutive *sections*. For example, if the first *section* starts at the hexadecimal address ”0x2000” and the following one at ”0x5000”, an offset of ”0x3000” has to be configured. The third *section* would start at address ”0x8000”. The number of *sections* are chosen by writing a number greater 1 to the register *Section Count*. The size of a single *section* is configured by using *Section Size*. When using more than one *section*, the configured size should not exceed the offset, since it would result in partially overwriting the preceding *section* with the following one. If the value of *Section Size* is less than *Section Offset*, the difference between two *sections* is skipped. Assuming the above example with a size of ”0x2200”, the address range from ”0x4200” to ”0x4fff” is skipped and therefore not read from memory. This method can be used for reading a sub-image from memory or skipping regular areas of a certain data layout. If `SUPPORT MULTIPLE SECTIONS` is not set, *Section Offset* and *Section Count* are ignored and a single *section* is assumed.

After starting the operation of the `as_memreader` for the first time by setting the `GO` signal, the module continuously provides the current physical memory address of the *Address Generator* to the software. This address is the one used for the next memory access but might not have been processed yet. It is stored within the hardware register `current hw addr`. At the end of the operation of the `as_memreader`, the address within this register points at the first address following the current *section* if `SUPPORT_MULTIPLE_SECTIONS` has been set to ”false”. Otherwise, it points at the start address of the following *section*, although it is not used. This is due to the fact, that the address is incremented by the *offset* after each *section*, where the last one is not handled separately. The address is only updated during the operation of the `as_memreader` and therefore is also not cleared once it has finished its operation. However, performing a *reset* on the module, either by hardware or software, clears the register.

The `as_memreader` preferably utilizes burst accesses to read a chunk of memory at once, since requesting the memory bus requires a considerable overhead. By using bursts, the number of bus accesses and therefor required overhead can be reduced. The number of bytes used for a burst access can be configured using the register *Maximal Burst Length*. The configured number of bytes written to this register has to be a multiple of the of the memory bus interface. Additionally, the maximal amount of data transferred in a single burst is usually limited by the architecture, to prevent other bus masters from starving, which must not be exceeded. If the configured *Section Size* is not a multiple of *Maximal Burst Length*, the as_memreader module falls back to less efficient single beat accesses for the remaining bytes, if SUPPORT_VARIABLE_BURST_LENGTH has not been set. The module requests the number of bytes equal to the one used for the memory bus interface at a time, which is usually 4 or 8 bytes. If SUPPORT_VARIABLE_BURST_LENGTH has been set, however, the as_memreader performs a burst access for equal to the remaining number of bytes.

The as_memreader module does not perform any kind of error handling (see Chapter 5.2). Since the module is designed to actively "pull" data, a SYNC_ERROR due to lost data cannot occur. Additionally, the as_memreader is not able to determine whether its received data posses a valid value and therefore does not generate a DATA_ERROR either.

Instead of directly utilizing a memory bus master, the as_memreader module can also be connected to an *as_arbiter* for mapping multiple *ASTERICS* memory modules to a single memory bus master. The as_memreader requests a bus access by setting the port mem_req to "1" and subsequently for the same value appearing at its mem_req_ack port. The latter port has to be bound to "1" (high), if no *as_arbiter* is used.

Optionally, the as_memreader generates a high level ("1") at its interrupt_out port when it finishes its configured data transfer operation, i.e. all *sections* have been read from memory. Therefor, the pre-synthesis parameters SUPPORT_INTERRUPTS and SUPPORT_DONE_IRQ_SOURCE have to be set to "true". The signal is active for a single clock cycle and can be used for generating a hardware interrupt for the processor, by using one of the available interrupt lines.

7.2.1.6. Module Driver

In order to set up data transfers from software, a module driver, namely as_reader_writer, is provided for the as_memreader. This driver is implemented in *C* and comprises a header and a source file. Within the module driver, a number of macros are defined for calculating the offset of the hardware registers and their bit indices. These macros can be used for interfacing the hardware registers of the as_memreader manually. Alternatively, the functions of the module driver can be used. The provided functions enable to utilize the functionality of the module without having to look up the appropriate macros. Internally, the ASTERICS *Support Library* is used for performing the actual accesses to hardware.

7.2.1.7. Application Notes

For common applications using as_memreader module, the default settings for most of its pre-synthesis parameters can be used. The only exception is DOUT_WIDTH and MEMORY_DATA_WIDTH which may be adjusted to 64 bit. The following Listing 7.1 shows how the as_memreader module is usually set up for a bare-metal application. For using the module with an operating system, the device driver of the *ASTERICS* framework has to be used (see Chapter 4.5). As a first step, the registers of the module are set by using the function as_reader_writer_init(). It takes two arguments, the start address of the module, which is defined in *as_hardware.h*, and a pointer to the configuration structure as_reader_writer_config_t defined in the header file of the module driver. By allocating a structure of this type, the corresponding data fields can be set in advance, to configure the module at once. If a NULL pointer is provided to `as_reader_writer_init()`, the default values defined in the header file are used, which covers most applications. The function also resets the hardware module. The *section address* and *section size* have to be manually configured by the user by using `as_reader_writer_set_section_addr()` and `as_reader_writer_set_section_size()` respectively. A default *section size* is set by `as_reader_writer_init()`, however, the value is likely to differ from what is required by the user. Lastly, `as_reader_writer_set_go()` starts the operation. For some applications it may be required to check whether the module has completed its operation by using `as_reader_writer_is_done()`. The given example shows an active status polling of the device, but different methods may also be used.

```c

/* Define a section size; here: the resolution of an image in
   byte */
#define IMAGE_RES           640*480

/* Allocate a memory area */
void *image_address = as_malloc(IMAGE_RES);


/******* Setting up the as_memreader module *******/

/* Sets default values for burst length, etc. */
as_reader_writer_init(AS_MODULE_BASEADDR_MEMREADER_0, NULL);

/* Set the start address, where the as_memreader is supposed to
   read from */
as_reader_writer_set_section_addr( \
    AS_MODULE_BASEADDR_MEMREADER_0, \
    (uint32_t*) image_address);

/* Set the number of bytes to be read */
as_reader_writer_set_section_size( \
    AS_MODULE_BASEADDR_MEMREADER_0, IMAGE_RES)


/************** Data transfer **************/

/* Start the as_memreader */
as_reader_writer_set_go(AS_MODULE_BASEADDR_MEMREADER_0);

/* Wait until the as_memreader has completed the section */
while(!as_reader_writer_is_done( \
    AS_MODULE_BASEADDR_MEMREADER_0)) {
/* Do nothing */
}


```

Listing 7.1: Using the `as_memreader` module.

7.2.2. as_memwriter

by Alexander Zoellner

7.2.2.1. Brief Description

The `as_memwriter` is a hardware module for efficiently transferring data from the FPGA to main memory. The module utilizes an *as_stream* interface to programmable logic and a bus master interface to memory. It transfers chunks of data , so called *sections*, by using burst accesses whenever possible. *Sections* are defined by a start address and size. Multiple *sections* are also supported, which may include a constant offset (i.e. the difference between the start address of two consecutive *sections*) in between, allowing to write rectangular sub-images or complying to a certain data layout. In order to decrease the amount of time spent idle, the following *section* can be programmed during an ongoing operation by overwriting the hardware registers of the `as_memwriter` and setting its `go` flag again. The module proceeds with the next data transfer after the current *section* has been completed, i.e. all data has been written. The module provides a register for tracking the progress of the current *section*, which holds the next address the module is going to read from. Additionally, the `as_memwriter` supports organizing data in *data units*, which is a logically grouped number of bytes. A counter, readable by software, is used for tracking the number of *data units*, which have been transferred to memory. This can be used for counting frames, image lines or any arbitrary data sets.

7.2.2.2. Architecture

Figure 7.2 shows the architecture of the `as_memwriter` module in simplified form to emphasize the main parts of the module, which are required for performing data transfers. The `as_memwriter` obtains its data via a hardware module, using the `as_stream` interface (see Chapter 5.3). The incoming data is intermediately stored in a *FIFO Buffer*, in order to aggregate a certain number of bytes for performing more efficient data transfers towards memory, using *bursts*. Further, data loss is prevented in this manner, if access to the memory bus cannot be obtained right away. If the *FIFO Buffer* reaches its limit, the `as_memwriter` requests the preceding module to suspend any further data transfers to its `as_stream` interface. The `as_memwriter` utilizes a *Master Interface* for transferring data to memory. This interface comprises only signals which are required for configuring the access to memory, such as the number of bytes to be transferred or when to start a data transfer. The actual memory bus access is performed by a *Bus Translation* module, which is connected to the memory bus interface and the `as_memwriter`. Occasionally, the user may not require the data of the image processing chain, which results in the `as_memwriter` to not transfer any data to memory. In this case, the *Enable Logic* is used to discard any following data at the `as_stream` interface, to prevent the *FIFO Buffer* from overflowing. Thus, the `as_memwriter` does not have to request the preceding module to suspend its data transfers, since other parts of the processing chain may depend on the data (e.g. the `as_memwriter` is only used to store intermediate results for control purposes). For regular operation, the *Enable Logic* merely forwards the data to the *FIFO Buffer*.

For configuring the `as_memwriter`, a *Register Interface* is utilized, which are associated with a number of hardware registers. These registers can be accessed by hardware and software alike, for exchanging control and status information. The control information are provided by software via the *Register Interface* to affect the behavior of the `as_memwriter`. For this reason, the appropriate hardware registers are connected to the *Enable Logic* to choose whether the `as_memwriter` is expected to discard data at its `as_stream` interface. The *Memory State Machine* is the central part of the module and is responsible for setting up data transfers, according to the settings of the software within the hardware registers. It also affects the *Enable Logic* under certain conditions.

During an ongoing operation, certain information must not be changed, in order for the `as_memwriter` module to operate correctly. For this reason, information regarding the specifics of the operation, such as number of *sections* or its size, have to be kept stable. This is accomplished by copying critical information to *Shadow Registers* at the start of an operation, which is determined by the *Memory State Machine*. As a side effect, this allows the software to replace the associated hardware registers during an ongoing operation, which is used for queuing the subsequent operation for transferring data. The *Address Generator* calculates the address required for the individual memory accesses as well as the number of bytes to be transferred. The required information are obtained from the *Shadow Registers*. Towards software, the address used for the next data transfer is published to software via the *Register Interface*.

Optionally, the *Data Unit Complete Logic* can be added to the `as_memwriter` if `SUPPORT_DATA_UNIT_COMPLETE` has been set. It is used for tracking the number of *data units* which have been transferred to memory. This number as well as the end address of the last *data unit* is published to software using the *Register Interface*. For certain configurations, the *Data Unit Complete Logic* also affects the *Enable Logic*.

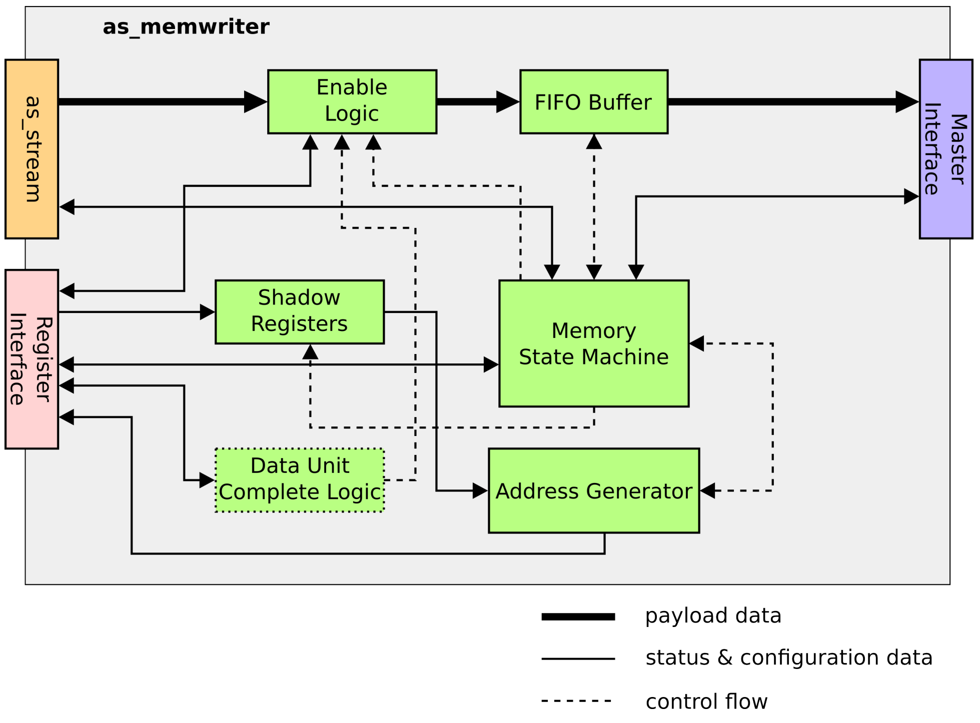

Figure 7.2.: Architecture of the `as_memwriter` module

7.2.2.3. Pre-Synthesis Options

Name	Range	Description

REGISTER_BIT_WIDTH	Positive integer value	Bit width for the slave registers used by this module (usually 32 bit).
DOUT_WIDTH	Positive integer value	Bit width of the data port of this module. Currently uses the same bit width as the memory bus interface (usually 32 or 64 bit).
MEMORY_DATA_WIDTH	Positive integer value	Bit width of the data port to memory. Depends on the setting of the memory bus (usually 32 or 64 bit).
MEM_ADDRESS_BIT_WIDTH	Positive integer value	Bit width of the address port of the memory bus. Depends on the memory bus interface (usually 32 bit).
BURST_LENGTH_BIT_WIDTH	Positive integer value	Bit width of the memory bus to configure the burst size (currently 12 bits).
MAX_PLATFORM_BURST_LENGTH	Positive integer value	Highest possible burst length support by the memory bus in bytes (currently: 256).
FIFO_NUMBER_OF_BURSTS	Positive integer value	Minimum size of the internal data fifo. The actual size is to the power of 2 required for fitting the chosen size.
SUPPORT_MULTIPLE_SECTIONS	Boolean	Enables regular address jumps for reading data from memory.
SUPPORT_INTERRUPTS	Boolean	Allows the module to generate interrupt events.

SUPPORT_DONE_IRQ_SOURCE	Boolean	Module generates an interrupt event at the end of a *section*. SUPPORT_INTERRUPTS has to be enabled.
SUPPORT_DUC_IRQ_SOURCE	Boolean	Module generates an interrupt event at the end of a data unit. SUPPORT_INTERRUPTS has to be enabled. SUPPORT_DATA_UNIT_COMPLETE has to be enabled.
SUPPORT_DATA_UNIT_COMPLETE	Boolean	Module generates an interrupt event at the end of a data unit. SUPPORT_INTERRUPTS has to be enabled.
UNIT_COUNTER_WIDTH	Positive integer value	Bit width of the register for storing the number of data units.

7.2.2.4. Register Space

Name	Relative Address	Width	Description
`state/control`	0x0	32	Status information of the `as_memwriter` and controlling the operation of the module. The lower half of the register is used for status information, the upper half for configuration.
`section addr`	0x4	32	Memory address at which the module starts to read data from.
`section offset`	0x8	32	Address offset in byte for regular address jumps. The offset is the distance between the start addresses of two consecutive *sections*.

`section size`	0xC	32	Size of a *section* in byte to be read from memory. The size has to be a multiple of the configured data bus width in byte.
`section count`	0x10	32	Number of *sections.* If `SUPPORT_MULTIPLE_SECTIONS` is not set, this register is ignored and a single *section* is assumed. Otherwise, an address jump is performed between two consecutive *sections.*
`max burst length`	0x14	32	Number of bytes to be requested for a single memory bus access. It has to be a multiple of `MEMORY_DATA_WIDTH` and must not exceed `MAX_PLATFORM_BURST_LENGTH`.
`current hw addr`	0x18	32	Contains the next memory address the module is going to read from. If the `as_memwriter` is not programmed this address is 0x0. For multiple *sections*, it points to the last *section* start address + offset after the last *section* has been served.
`last data unit complete addr`	0x1C	32	Following address after a data unit has been written to memory. `SUPPORT_DATA_UNIT_COMPLETE` has to be set. Otherwise the value of this register is always 0x0.
`current unit count`	0x20	32	Number of transferred data units to memory. `SUPPORT_DATA_UNIT_COMPLETE` has to be set. Otherwise the value of this register is always 0x0. The actual number of utilized bits of this register depend on the setting of `UNIT_COUNTER_WITDH`, up to 32.

Table 7.11.: Register overview of the `as_memwriter` module.

Field Name	**Bit Index**	**Type**	**Reset Value**	**Description**

`flush data`	23	wo	0x0	If set, the `as_memwriter` to transfer all currently buffered data to memory.
`disable on no go`	22	wo	0x0	If set, the `as_memwriter` to reject data at its input port after finishing its current operation and the next one has not been set up (i.e. there is no `pending go`).
`single shot`	21	wo	0x0	If set, the `as_memwriter` only transfers a single *data unit* before it continuous to reject incoming data.
`enable on data unit complete`	20	wo	0x0	If set, the `as_memwriter` starts accepting data at its input port after receiving a signal at its `data_unit_complete_in` port.
`disable`	19	wo	0x0	If set, the `as_memwriter` rejects all incoming data.
`enable`	18	wo	0x0	If set, the `as_memwriter` accepts incoming data.
`go`	17	wo	0x0	If set, the `as_memwriter` starts transferring data to memory.
`reset`	16	wo	0x0	Resets the module and clears all remaining data in the *FIFO Buffer*.
`set enable`	6	ro	0x0	If set, the `as_memwriter` currently accepts incoming data.
`pending go`	5	ro	0x0	If set, the next data transfer has already been programmed.

`flushable data`	4	ro	0x0	If set, there is currently data in the *FIFO Buffer*, which can be flushed.
`sync error`	3	ro	0x0	If set, data has been lost due to the preceding modules ignoring the `STALL` signal and the *FIFO Buffer* being full.
`busy`	1	ro	0x0	The `as_memwriter` is currently operating.
`ready`	0	ro	0x0	The as_memwriter is currently idle.

Table 7.12.: Bit field overview of the combined `state/control` register of the `as_memwriter`.

Field Name	Bit Index	Type	Reset Value	Description
`section address`	31:0	wo	0x0	Start address for reading data from memory.

Table 7.13.: Bit field overview of the `section addr` register of the `as_memwriter`.

Field Name	Bit Index	Type	Reset Value	Description
`section offet`	31:0	wo	0x0	The offset is the distance between two start addresses in byte.

Table 7.14.: Bit field overview of the `section offset` register of the `as_memwriter`.

Field Name	Bit Index	Type	Reset Value	Description
`section size`	31:0	wo	0x0	Size of a section in byte to be read from memory.

Table 7.15.: Bit field overview of the `section size` register of the `as_memwriter`.

Field Name	Bit Index	Type	Reset Value	Description
`section count`	31:0	wo	0x0	Number of sections.

Table 7.16.: Bit field overview of the `section count` register of the `as_memwriter`.

Field Name	Bit Index	Type	Reset Value	Description
`max burst length`	31:0	wo	0x0	Number of bytes to be requested for a single memory bus access.

Table 7.17.: Bit field overview of the `max burst length` register of the `as_memwriter`.

Field Name	Bit Index	Type	Reset Value	Description
`current hw addr`	31:0	ro	0x0	Next address the `as_memwriter` is going to write to.

Table 7.18.: Bit field overview of the `current hw addr` register of the `as_memwriter`.

Field Name	Bit Index	Type	Reset Value	Description
`reg last data unit complete addr`	31:0	ro	0x0	Following address after the last *data unit*. `SUPPORT_DATA_UNIT_COMPLETE` has to be set.

Table 7.19.: Bit field overview of the `reg last data unit complete addr` register of the `as_memwriter`.

Field Name	Bit Index	Type	Reset Value	Description
`current unit count`	31:0	ro	0x0	Number of *data units* which have been written to memory. `SUPPORT_DATA_UNIT_COMPLETE` has to be set.

Table 7.20.: Bit field overview of the `current unit count` register of the `as_memwriter`.

7.2.2.5. Behavior

The bit fields of the `control` and `state` registers are also represented internally within the module by signals with the same name. When referring to the bit field, the name is written in lower case and in upper case for the signal. Flags refer to the bit fields.

When the module receives a reset request, either by using the hardware port or the `reset` field of the `control` register, ongoing bus accesses are terminated and all internally stored configuration settings are cleared after a single clock cycle. Additionally, the module requests to clear the `go`, `enable`, `disable` and `reset` field. The configuration registers towards software are left as is. Simultaneously, the data in the *FIFO Buffer* is discarded and the *Address Generator* is reset. Further, the *Enable Logic* is instructed to discard all incoming data at the `as_stream` interface. The reset persists only for one clock cycle if it has been requested by software, otherwise up to the point it is lifted by hardware. During the reset, neither of the modules' status flags is set. After the reset has been lifted, the `as_memwriter` assumes its idle state.

In the modules' idle state, the `ready` flag is set, the `busy` flag is unset and the configuration registers are continuously copied into internal *Shadow Registers*. Upon receiving the `GO` signal, the current configuration is used for setting up data transfers after one clock cycle. Further, the `ready` flag is unset and the `busy` flag is set instead, which is kept throughout the whole operation. The `as_memwriter` requests the slave logic to clear the `go` bit field and starts setting up data transfer using the *Master Interface*, as soon as enough data is aggregated in the *FIFO Buffer*. The required number of bytes for the data transfer as well as the type of transfer (single-beat or burst) is determined by the *Address Generator*. The `as_memwriter` uses the setting of the *Address Generator*, except a *flush* request occurs. During operation, the configuration registers at the *Register Interface* can be overwritten and the `go` bit field set again, to seamlessly queue the next data transfer. If the `go` flag was set in this manner, the module sets its `PENDING GO` signal. Since the `as_memwriter` copies the configuration only at the start, it would be possible to change the configuration for the following transfer after having already set the `go` field again. However, it is strongly recommended to configure the data transfer before queuing the next transmission. If `disable on no go` field of the `control` register has been set and the `GO` signal is not present at the time the `as_memwriter` finishes its operation, it requests the *Enable Logic* to discard incoming data. Further, the *FIFO Buffer* is reset. Subsequently, the `as_memwriter` returns to its idle state. Otherwise, the module sets its `ready` flag before returning to its idle state. Here, the `busy` and `ready` flag are set simultaneously for a single clock cycle, which is exclusively used if another hardware module interfaces the `as_memwriter` instead of software. This is mainly for backwards compatibility and may be subject to change in the future. If the next `GO` has already been set when entering the idle state, the module proceeds in the aforementioned manner without delay. The `PENDING GO` (since `GO` is unset) is unset again.

The in- and output side of the `as_memwriter` are decoupled from each other. At its output side, data can be written to memory as long as there is remaining data within the *FIFO Buffer*. The data output is started once the `as_memwriter` has been programmed and the `GO` signal has been set. However, in order to accept data at its input side, the *Enable Logic* has to set the internal `ENABLE` signal. This signal is combined with the `STROBE` signal, using an "AND" operation. The setting of the `ENABLE` signal can be influenced directly and indirectly by software using the appropriate fields of the `control` register. The `ENABLE` signal can be set and unset directly using the `enable` and `disable` field of the `control` register, respectively. These fields affect the setting of the *ENABLE* signal directly after a single clock cycle. If both fields are set simultaneously, the `disable` is dominant. The `ENABLE` signal can be set for or after the `go` field has been set. The user is responsible for preventing data loss manually when using the aforementioned fields directly during the operation of the `as_memwriter`. On the other hand, the `disable on no go` field can be used for or during the operation of the module to unset the `ENABLE` signal indirectly. The *Memory State Machine* of the as_memwriter evaluates this field upon finishing its operation. If disable on no go is set and PENDING GO is not set, the ENABLE signal is unset. If a PENDING GO is present, disable on no go takes no effect but is evaluated again for subsequent operations. If the *Data Unit Complete Logic* is active by having set the SUPPORT_DATA_UNIT_COMPLETE pre-synthesis parameter, additional options for influencing the *Enable Logic* are available. For setting the ENABLE signal, the enable on data unit complete field is used. The *Data Unit Complete Logic* of the as_memwriter sets the ENABLE signal on its own, when it receives the next DATA UNIT COMPLETE signal at its data_unit_complete_in port. This can be used to synchronize the as_memwriter with a continuous data stream. In order operate correctly, the ENABLE signal has to be unset before using the enable on data unit complete field and the *FIFO Buffer* is expected to not hold any data at this point. The single shot field can be used in combination with enable on data unit complete to have the as_memwriter automatically unset the ENABLE signal after receiving a subsequent DATA UNIT COMPLETE signal. If the ENABLE has been unset in this way, a snapshot is taken of the current fill-level of the *FIFO Buffer*. The *Data Unit Complete Logic* triggers an internal *flush* process for transferring all remaining data of the *data unit* to memory. The snapshot is used for determining the required number of bytes to be transferred. After the completion of this task, the *Memory State Machine* and *FIFO Buffer* are reset

The as_memwriter module is a *stall-generating* module (see Chapter 5.3.4). The module sets the signal for its stall_out port to high if either its *FIFO Buffer* is about to be full or during a *flush* process. For the former occasion, the signal is set if the *FIFO Buffer* has only one remaining free data entry. This guarantees that at least one additional data set is accepted after raising the STALL signal. The STALL is lifted as soon as there are two free entries within the *FIFO Buffer*. During a *flush* process, the as_memwriter also sets its STALL signal for as long as the operation is carried out. The *flush* ends with either the *FIFO Buffer* being empty or in case the next operation has not been programmed if the *flush* exceeds the currently programmed *sections*.

The as_memwriter module organizes its data transfers in one or more *sections*, where each one is a physically continuous chunk of data in memory. If more than one *section* is required, the configuration option SUPPORT_MULTIPLE_SECTIONS has to be set at synthesis time. The register *Section Address* defines the memory address of the first *section*, whereas *Section Offset* is used for the following ones. The latter is the difference of the start addresses of two consecutive sections. For example, if the first *section* starts at the hexadecimal address ”0x2000” and the following one at ”0x5000”, an offset of ”0x3000” has to be configured. The third *section* would start at address ”0x8000”. The number of *sections* are chosen by writing a number greater 1 to the register *Section Count*. The size of a single *section* is configured by using *Section Size*. When using more than one *section*, the configured size should not exceed the offset, since it would result in partially overwriting the preceding *section* with the following one. If the value of *Section Size* is less than *Section Offset*, the difference between two *sections* is skipped. Assuming the above example with a size of ”0x2200”, the address range from ”0x4200” to ”0x4fff” is skipped and therefore not read from memory. This method can be used for writing a sub-image to memory or complying to a certain data layout. If SUPPORT MULTIPLE SECTIONS is not set, *Section Offset* and *Section Count* are ignored and a single *section* is assumed.

After starting the operation of the as_memwriter for the first time by setting the GO signal, the module continuously provides the current physical memory address of the *Address*

Generator to the software. This address is the one used for the next memory access but might not have been processed yet. It is stored within the hardware register `current hw addr`. At the end of the operation of the `as_memwriter`, the address within this register points at the first address following the current *section* if `SUPPORT_MULTIPLE_SECTIONS` has been set to "false". Otherwise, it points at the start address of the following *section*, although it is not used. This is due to the fact, that the address is incremented by the *offset* after each *section*, where the last one is not handled separately. The address is only updated during the operation of the `as_memwriter` and therefore is also not cleared once it has finished its operation. However, performing a *reset* on the module, either by hardware or software, clears the register.

When the `SUPPORT_DATA_UNIT_COMPLETE` parameter is set, the `as_memwriter` counts the number of *data units* which have been transferred to memory. Each time a `DATA UNIT COMPLETE` signal is received, a snapshot of the current fill-level of the *FIFO Buffer* is taken, to determine the remaining data associated with the last *data unit*. After having transferred the data to memory, the `current unit counter` register is updated, by incrementing its value by one.

Similarly, the `last data unit complete addr` is also updated by writing the following address of the *data unit* to it. As an example, if the start address of the *data unit* is "0x0000" and its size is "0xff", the resulting address within the `last data unit complete addr` register is "0x0100", because it is the next following address of the *data unit*.

Since the `as_memwriter` receives data from another hardware module, it comprises a `SYNC_ERROR` signal (see Chapter 5.2). This signal is set when data is lost at the `as_memwriter`, due to receiving data despite its *FIFO Buffer* being full. It is propagated to hardware using the `sync_error_out` port. The software can also check if this signal is set, by reading the `sync error` field of the `status` register. Once the `sync_error` is set, it persists until the `as_memwriter` is reset.

Instead of directly utilizing a memory bus master, the `as_memwriter` module can also be connected to an *as_arbiter* for mapping multiple *ASTERICS* memory modules to a single memory bus master. The `as_memwriter` requests a bus access by setting the port `mem_req` to "1" and subsequently for the same value appearing at its `mem_req_ack` port. The latter port has to be bound to "1" (high), if no *as_arbiter* is used.

Optionally, the `as_memwriter` generates a high level ("1") at its `interrupt_out` port when it finishes its configured data transfer operation, i.e. all *sections* have been read from memory. Therefor, the pre-synthesis parameters `SUPPORT_INTERRUPTS` and `SUPPORT_DONE_IRQ_SOURCE` have to be set to "true". If SUPPORT_DATA_UNIT_COMPLETE is set, the `SUPPORT_DUC_IRQ_SOURCE` parameter can be set to generate an interrupt event after writing a *data unit* to memory. The signals for the `interrupt_out` port are active for a single clock cycle and can be used for generating a hardware interrupt for the processor, by using one of the available interrupt lines.

The `as_memwriter` offers a *flush* mechanic to force writing all intermediately stored data within the *FIFO Buffer* to memory. This process can either be triggered externally by hardware using the `flush_in` port or by software using the `flush data` field of the `control` register. The former is one of the common module signals (see Chapter 5.2), whereas the latter is part of the *Register Interface*. Either signal is expected to hold a "high" ("1") for a single clock cycle but it is also valid to extend this signal to multiple clock cycles. When the `as_memwriter` receives a *flush* request by software, it signals the module connected to its *Register Interface* to unset the corresponding field of the `control` register. Towards hardware, no such signal is used. After one clock cycle, the *flush* request is adopted by the `as_memwriter`. The module sets its `STALL` signal and starts transferring data to memory, as soon as the `as_memwriter` is in its operation mode, i.e. the `GO` signal has been set. The `as_memwriter` concludes its *flush* process, if either its *FIFO Buffer* is empty or the `DONE` signal is set internally and there is no `PENDING GO`, i.e. the *idle* state is assumed and the next operation has not been set up. Thus, if the `as_memwriter` is not executing an operation, i.e. `go` has not been set, the *flush* is concluded immediately, without setting the `STALL` signal at all. If the amount of data within the *FIFO Buffer* exceeds the remaining amount of the currently configured *sections*, the `PENDING GO` signal is checked. Similar to the double buffering scheme for queuing operations (`PENDING GO`), the *flush* process can also be queued across operations. If the `PENDING GO` is absent at the end of the operation, the *flush* process has to be triggered anew. This scheme prevents the `STALL` signal of the `as_memwriter` to be set for an extended period of time, due to not setting up the following operation right away. Further, the source of the *flush* request (hardware or software) is not required to check the current status of the `as_memwriter`, whether further action is required. The `as_memwriter` uses the setting of the *Address Generator* for determining whether *burst* data transfers are to be used, as long as the amount of data within the *FIFO Buffer* permits it. Otherwise a maximum of `max_burst_length`$*2 - 2$ *single-beat* transfers are used. This is the worst-case scenario, which occurs if the *flush* process involves two data transfer operations, where the first one has to be concluded first before executing the second one. Usually, the expected number of *single-beat* transfers is less than half of the worst-case assumption.

7.2.2.6. Module Drivers

In order to set up data transfers from software, a module driver, namely `as_reader_writer`, is provided for the `as_memwriter`. This driver is implemented in *C* and comprises a header and a source file. Within the module driver, a number of macros are defined for calculating the offset of the hardware registers and their bit indices. These macros can be used for interfacing the hardware registers of the `as_memwriter` manually. Alternatively, the functions of the module driver can be used. The provided functions enable to utilize the functionality of the module without having to look up the appropriate macros. Internally, the ASTERICS *Support Library* is used for performing the actual accesses to hardware.

7.2.2.7. Application Notes

For common applications using `as_memreader` module, the default settings for most of its pre-synthesis parameters can be used. The only exception is `DOUT_WIDTH` and `MEMORY_DATA_WIDTH` which may be adjusted to 64 bit. The following Listing 7.2 shows how the `as_memwriter` module is usually set up for a bare-metal application. For using the module with an operating system, the device driver of the *ASTERICS* framework has to be used (see Chapter 4.5). As a first step, the registers of the module are set by using the function `as_reader_writer_init()`. It takes two arguments, the start address of the module, which is defined in *as_hardware.h*, and a pointer to the configuration structure `as_reader_writer_config_t` defined in the header file of the module driver. By allocating a structure of this type, the corresponding data fields can be set in advance, to configure the module at once. If a NULL pointer is provided to `as_reader_writer_init()`, the default values defined in the header file are used, which covers most applications. The function also resets the hardware module. The *section address* and *section size* have to be manually configured by the user by using `as_reader_writer_set_section_addr()` and `as_reader_writer_set_section_size()` respectively. A default *section size* is set by `as_reader_writer_init()`, however, the value is likely to differ from what is required by the user. Lastly, `as_reader_writer_set_go()` starts the operation and `as_writer_set_enable()` allows the *FIFO Buffer* to accept data. Either sequence for setting the `go` and `enable` field can be chosen. For some applications it may be required to check whether the module has completed its operation by using `as_reader_writer_is_done()`. The given example shows an active status polling of the device, but different methods may also be used.

```c

    /* Define a section size; here: the resolution of an image in byte */
    #define IMAGE_RES           640*480

    /* Allocate a memory area */
    void *image_address = as_malloc(IMAGE_RES);


    /******* Setting up the as_memwriter module *******/

    /* Sets default values for burst length, etc. */
    as_reader_writer_init(AS_MODULE_BASEADDR_MEMWRITER_0, NULL);

    /* Set the start address, where the as_memreader is supposed to read from */
    as_reader_writer_set_section_addr( \
        AS_MODULE_BASEADDR_MEMWRITER_0, (uint32_t*) image_address);

    /* Set the number of bytes to be read */
    as_reader_writer_set_section_size( \
        AS_MODULE_BASEADDR_MEMWRITER_0, IMAGE_RES)


    /************** Data transfer **************/

    /* Start the as_memreader */
    as_reader_writer_set_go(AS_MODULE_BASEADDR_MEMWRITER_0);

    /* Enable input to FIFO Buffer */
    as_writer_set_enable(AS_MODULE_BASEADDR_MEMWRITER_0);

    /* Wait until the as_memreader has completed the section */
    while(!as_reader_writer_is_done(AS_MODULE_BASEADDR_MEMWRITER_0)) {
    /* Do nothing */
    }

    /* Disable input to FIFO Buffer */
    as_writer_set_disable(AS_MODULE_BASEADDR_MEMWRITER_0);


```

Listing 7.2: Using the `as_memwriter` module for a single *section*.

Listing 7.3 shows the how the `as_memwriter` is configured for the *single shot* mode to transfer exactly one *data unit* to memory. The aforementioned setup has been also used for this application. Contrary to the regular operation mode, the `enable` field is not set, since the `ENABLE` signal is set implicitly. Rather, the input of the *FIFO Buffer* is activated automatically by the `as_memwriter` upon receiving a "1" at its `data_unit_complete_in` port. For this reason, the `enable on data unit complete` register field is set, using the corresponding function. The `single shot` field is used by the `as_memwriter` to unset its `ENABLE` signal, once the following `DATA UNIT COMPLETE` signal has been received. The `as_memwriter` clears the `enable on data unit complete` and `single shot control` register field at the end of the *data unit*. Further, the *Memory State Machine* is reset, which terminates the current and pending operations.

Any sequence for setting the three required bit fields for the *single shot* mode may be chosen.

```c

    /******* Setting up the as_memwriter module *******/

    /* ... */


    /************** Data transfer *************/

    /* Start the as_memreader */
    as_reader_writer_set_go(AS_MODULE_BASEADDR_MEMWRITER_0);

    /* Activate single shot mode */
    as_writer_set_single_shot(AS_MODULE_BASEADDR_MEMWRITER_0);

    /* Automatically activate FIFO Buffer input */
    as_writer_set_enable_on_data_unit_complete( \
        AS_MODULE_BASEADDR_MEMWRITER_0);

    /* Wait until the as_memreader has completed the section */
    while(!as_reader_writer_is_done( \
        AS_MODULE_BASEADDR_MEMWRITER_0)) {
    /* Do nothing */
    }


```

Listing 7.3: Using *single shot* mode of the `as_memwriter` module.

7.3. as_memio

by Alexander Zöllner

7.3.1. Brief Description

The `as_memio` module offers a way for conveniently transferring data between the *ASTERICS* -based processing chain on hardware and the application software of the user. Unlike most modules of the *ASTERICS* framework, `as_memio` (memory input/output) is a pure software module and therefore does not provide a hardware counterpart on its own.

Rather, it utilizes the memory modules (`as_memreader`/`as_memwriter`) implemented in hardware and their corresponding drivers. Towards the user application software, POSIX-like interfaces are provided, which comprise the commonly utilized file operations, most users are familiar with from operating systems. The `as_memio` module has been designed for being operable with bare-metal applications as well as for being seamlessly integrated in a device driver for an operating system.

7.3.2. Architecture

Figure 7.3 shows the main components of the `as_memio` module and its relation to other hardware and software parts of *ASTERICS* . The module consists of a *Ring Buffer* for intermediately storing the data to be transferred. This buffer is a chunk of memory which has a physically concurrent address space. The data source writes to the *Ring Buffer* in a linear manner. When the end of the buffer is reached, it starts at the beginning of the buffer again. The data sink reads from the *Ring Buffer* in a similar manner. The *Memio File* represents a specific instance of the `as_memio` module, which is associated with a single memory module. The *Memory Module Settings* part of the *Memio File* contains the static configuration for the memory module, such as the burst length to be used (see Chapter 7.2.1/7.2.2). The *Buffer Handler* is responsible for managing accesses to the *Ring Buffer* to prevent buffer over- and underflows. Further, the dynamic configuration for the memory module is performed by this part. Status information of the memory module are used to take appropriate actions within the *Buffer Handler*. The `as_memio` module utilizes the `as_reader_writer` module driver for accessing the hardware memory module. Appropriate functions of the driver are chosen depending on the type of the associated memory module. Towards software, a range of interfaces are presented for conveniently transferring data between hardware and software. A new *Memio File* along with its associated *Ring Buffer* is created by using *open* and destroyed by *close*. Since a given instance of `as_memio` posses only one *Ring Buffer*, it can manage either a `as_memreader` or `as_memwriter` module. The presented figure shows an instance of `as_memio` utilizing a `as_memwriter`. The `as_memwriter` is the data source and writes its data to the *Ring Buffer*. The *read* interface is used by the *application* software for obtaining the data of the hardware. The *Buffer Handler* copies the requested data from the *Ring Buffer* to the *User Buffer*. In this case, the *write* interface is not available. When using a `as_memreader`, the data flow is inversed and the *read* interface becomes unavailable for the `as_memio` instance. Here, data is copied from the *User Buffer* to the *Ring Buffer*.

The *hw update* interface is used for explicitly triggering the *Buffer Handler* to prevent data from being ”stuck” within the *Ring Buffer*. The parameters required for the `as_memio` interfaces along with their behavior are presented in more detail in Chapter 7.3.4.

7.3.3. Compile-Time Options

Name	Range	Description

AS_MEMIO_DEFAULT_INTERFACE_WIDTH	Positive integer value	Sets the default bit width for ALL (!) memory module bus interfaces (usually 32 or 64 bit). This parameter is likely to be dropped in a future version of `as_memio`.
AS_MEMIO_DEFAULT_MAX_BURST_LENGTH	Positive integer value	Sets the default burst length in byte for ALL (!) memory modules (usually 256). This parameter is likely to be dropped in a future version of `as_memio`.
AS_MEMIO_DEFAULT_HW_TRANSFER_SIZE	Positive integer value	Sets the default *transfer size* in byte for ALL (!) `as_memio` modules. The *transfer size* is the minimum *section size* to be configured for the `as_memwriter`. Prevents data loss at the *FIFO Buffer* which may occur due to too small *sections*.
AS_MEMIO_DEFAULT_FIFO_BUFFER_SIZE	Positive integer value	Sets the default *Ring Buffer* size in byte for ALL (!) `as_memio` modules. Increasing the size allows the memory module to transfer more data for a single configuration.

7.3.3.1. Register Space

The module described does not contain any memory-mapped control or status registers, since it is a software module.

7.3.4. Behavior

In order to establish a connection between a memory module and `as_memio`, its *open* function has to be called, named `as_memio_open`. Here, the address of the memory module has to be provided, which is internally used for calls to the functions of the

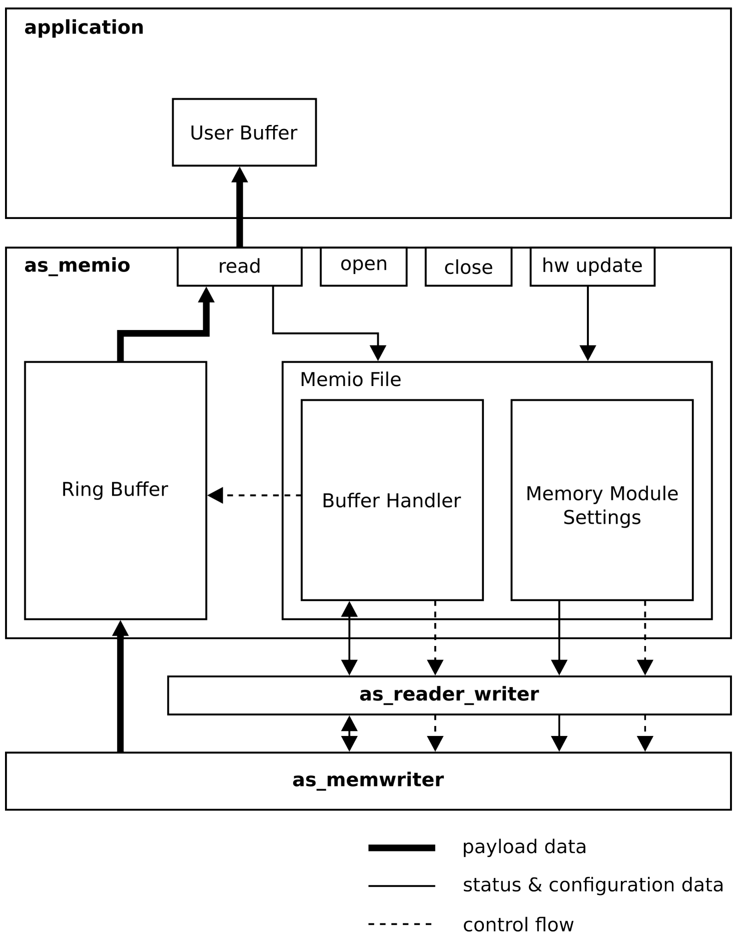

Figure 7.3.: Architecture of the `as_memio` module and its software/hardware interaction

`as_reader_writer` driver. By providing the direction of the data flow via the `flags` parameter, either a `as_memreader` or `as_memwriter` is associated with the instance of `as_memio`. The `as_memio` module is not able to determine the module type based on the address of the memory module. Alternatively, a pointer to a configuration structure of the type `as_memio_config_t` can be provided to overwrite the default settings for this `as_memio` instance (see 7.3.3). If a NULL pointer is provided, the default settings are used instead. Within the `as_memio_open` function, the *Ring Buffer* and *Memio File* are allocated. Subsequently, the static settings of the associated memory module are initialized, such as the `max burst length`. Lastly, the memory module is reset and the pointer to the *Memio File* is returned to the user for referencing to this instance of `as_memio`, similar to a file pointer of the POSIX *open* function. The contents of the *Memio File* are hidden from the user.

The function `as_memio_read` is used for transferring data from hardware to software. Next to the *Memio File* pointer, a *User Buffer* and the desired number of bytes have to be provided. The requested number of bytes are copied to this buffer and therefore has to be of appropriate size. First, the `as_memio_read` function is checks the current status of the `as_memwriter` module. The `current hw addr` is read, which determines the current location within the *Ring Buffer*, where the `as_memwriter` writes to. Additionally, if the `pending go` bit field of the `control` register is not set, the next *section* is programmed, as long as there is enough empty space within the *Ring Buffer*. All addresses up to `current` `hw addr` have been served by the hardware module. The actual amount of data available in the *Ring Buffer* is determined by using this address in combination with the last address copied to a *User Buffer* for a previous call to `as_memio_read`. Figure 7.4 shows how the available data within the *Ring Buffer* is determined. The `current hw addr` is shown as *hw* and the following address after the last copy process to the *User Buffer* as *sw*. Both are represented within the *Buffer Handler* of `as_memio`. If the number of bytes in the *Ring Buffer* is not equal to the requested one, the smaller number of bytes is copied. The address of *sw* is increased for each copied byte to the *User Buffer* throughout the lifetime of the `as_memio` instance. If either *hw* or *sw* exceeds the upper boundary of the *Ring Buffer*, it is set to its start address again (bottom). The *Buffer Handler* prevents buffer over- and underflow due to one pointer overtaking the other. After completing `as_memio_read`, the actual number of copied bytes is returned to the caller and the *User Buffer* contains the data.

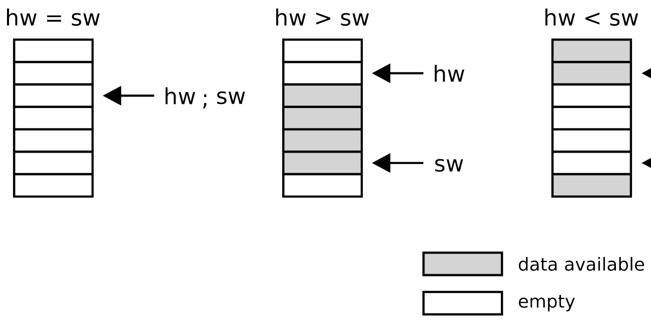

Figure 7.4.: Determining the available amount of data within the *Ring Buffer* for `as_memio_read`

The `as_memio_write` function operates in a similar way but data is copied from the *User Buffer* to the empty slots of the *Ring Buffer*. Figure 7.5 shows how the available addresses within the *Ring Buffer* are determined. Since a `as_memreader` is associated with this function it can only be configured once data is available in the *Ring Buffer*. Therefore, `current hw addr` is read after copying the data from the *User Buffer* as well as programming the next *section* in case the `pending go` field of the `control` register is not set. Although `as_memio` has been able to copy all data to the *Ring Buffer*, the actual data transfer may not yet be completed after `as_memio_write` returns.

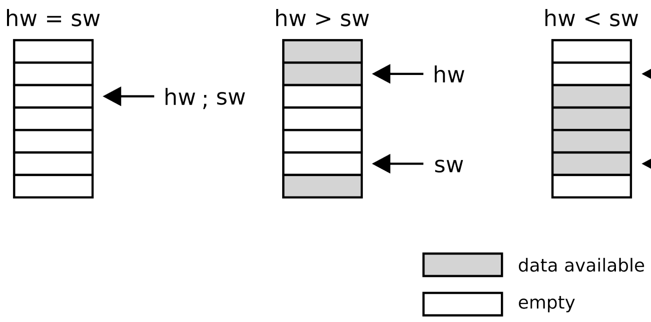

Figure 7.5.: Determining the available amount of data within the *Ring Buffer* for `as_memio_write`

The `as_memio` module implicitly programs its associated memory module when calling either `as_memio_read` or `as_memio_write`. Although sufficient in most cases, explicitly programming the memory module is required if the last data transfer between the *Ring Buffer* and *User Buffer* results in a boundary crossing by the hardware module. Since the memory modules only support a single *section* which has to consist of physically concurrent addresses, wrapping around the *Ring Buffer* requires programming the memory modules twice. This is shown in Figure 7.4 and 7.5 for ”hw > sw”, where the first *section* has to be programmed starting from *hw* up to the end of the *Ring Buffer* and the second on from the beginning. The second *section* can be programmed by using the `as_memio_hw_update` function. When the instance of `as_memio` is associated with a `as_memreader`, it prevents data from getting ”stuck” in the *Ring Buffer* which could cause the image processing chain to wait indefinitely for data to arrive. For the `as_memwriter`, it prevents a potential overflow of its *FIFO Buffer* due to not being able to transfer the data to the *Ring Buffer*.

Although the `as_memwriter` is able to request the preceding hardware module to suspend data transfers when its *FIFO Buffer* is full, not all modules are able to suspend their transfers (e.g. camera). For this reason, the `as_memio` module uses a *transfer size* when programming the *section* size for the `as_memwriter` to cater for continuous data streams. The *transfer size* is the minimum number of bytes which have to be transferred by the `as_memwriter` to prevent overflows of its *FIFO Buffer* due to setting up a too small *section*. This is mainly relevant when the `as_memwriter` is about to cross the upper boundary of the *Ring Buffer* of the `as_memio` module. Since two *sections* have to be programmed when wrapping around the *Ring Buffer*, the first one has to be big enough to give the software time to program the second one. The *FIFO Buffer* of the `as_memwriter` must not overflow during this given time window. The *Ring Buffer* of `as_memio` has to be a multiple of the *transfer size*.

The call to `as_memio_close` deletes a no longer required instance of the `as_memio` module. Here, the associated memory module is reset and all acquired resources are returned. Since the *Memio File* no longer exists after this point, it can no longer be used.

All calls to the functions of the `as_memio` module are nonblocking and therefore return immediately even if the request could not be fulfilled. This may require to call `as_memio_read` or `as_memio_write` more than once to transfer the desired number of bytes.

7.3.5. Application Notes

Figure 7.4 shows the setup of the `as_memio` module using default settings.

```

#define IMAGE_RES           640*480
int n;

void *user_buffer = as_malloc(IMAGE_RES);

/******* Setting up as_memio *******/

struct as_memio_file_s *memio_read_fp = \
    as_memio_open(AS_ADDR(AS_MODULE_BASEREG_MEMWRITER_0), \
        NULL, O_WRONLY);


/************** Data transfer **************/

```

```
    n = 0;

    while(n < IMAGE_RES) {
    n += as_memio_read(memio_read_fp, user_buffer+n, IMAGE_RES-n)
    }


```

Listing 7.4: Using `as_memio` for transferring data from hardware to software

7.4. Converters and Adapters

7.4.1. as_collect

by Julian Sarcher, Alexander Zöllner

7.4.1.1. Brief Description

The collect module collects smaller data words (e.g. 8 bit wide) until a certain bit width is reached. As the desired bit width is reached, the collect module forwards the larger data word. An example usage for this module is collecting 8-bit gray scale pixels for 32 or 64 bit memory bus accesses.

7.4.1.2. Configuration Options

Name	Description	Range
DIN_WIDTH	Data width of DATA_IN	Power of two $\wedge$ DIN_WIDTH $<$ DOUT_WIDTH
DOUT_WIDTH	Data width of DATA_OUT	Power of two $\wedge$ DOUT_WIDTH $<$ DIN_WIDTH

7.4.1.3. Register Space

The module described does not contain any memory-mapped control or status registers.

7.4.1.4. Resource Utilization

DIN_WIDTH	DOUT_WIDTH	Slices	Block-RAM	DSP-Slices
8	64	26 (0.2%)	0 (0.0%)	(0.0%)

7.5. 2D Window Modules

`- This section is currently under construction -`

7.6. *i2c* Bus Master

by Philip Manke

This section describes the module as_iic in detail. as_iic implements a simple *i2c* master with relatively limited functionality.

7.6.1. General Overview of Features and Limitations

The entire module was developed with a mindset of keeping the hardware small and adding only the most important features. Its primary use-case is configuring the cameras that are used with *ASTERICS* .

Figure 7.6 gives a general overview of the ports of the module.

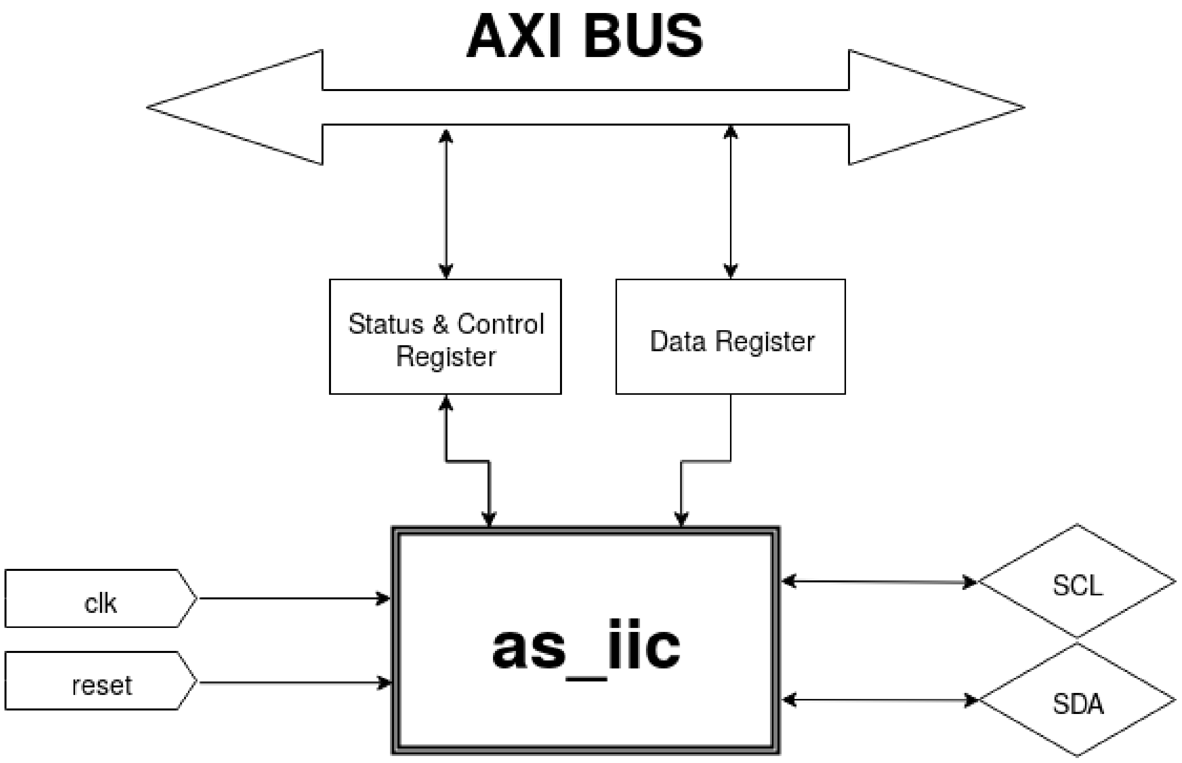

Figure 7.6.: Interfaces of the as_iic hardware module

The following features are supported:

- *standard* and *fast mode i2c.*
- Variable bus clock (configurable via software) of frequencies between 10 kHz to 1 MHz
- Clock stretching
- 7 bit addressing
- Multi-byte transactions

- Master-acknowledge

The modules limitations are:

- Only 8 bit addressing.
- No arbitration of any kind. Therefore, multi-master configurations are not supported.
- No faster modes of operation than Fast Mode.
- No interrupt support. The driver utilizes polling.
- No buffering of data. The driver has to transfer each byte sequentially.

10 bit addressing is possible with the current hardware, though it hasn't been implemented by the driver.

7.6.2. The Hardware

The hardware core consists of two state machines: one handles the generation of the bus clock (SCL) and the other controls the data signal (SDA) and the behaviour of the entire module.

7.6.2.1. The interfaces of `as_iic`

The `as_iic` module requires two 32 bit slave registers on an AXI-Slave bus. The first register is a combined status and control register, which also transfers the data received from the *i2c* slave. The second register is read-only for the `as_iic` hardware and is used to transfer the data to send to the *i2c* bus and for configuring the *i2c* bus clock frequency.

The status and control register is provided as three 16 bit registers for the status bits, the control bits and a special control-reset register respectively. The control-reset register allows the AXI-Bus to reset control bits by itself. If a bit in the control-reset register goes high, the respective bit in the control register is automatically set to low. Table 7.22 gives an overview of the registers of `as_iic`.

Register Name	Access	Offset	Description
`Status & Control Register`	RW	1	Status and control register for the hardware Half of this register reports the current status of the hardware module of `as_iis` to the software. It is also used to transfer bytes, received from slaves on the *i2c* bus, to the software. The other half is used to control the hardware.

Register Name	Access	Offset	Description
`Data Register`	W	0	Data register for the hardware This register is used to control the hardware module. Transactions can be initiated, stopped and modified, using bits of this register. It also contains a soft reset bit.

Table 7.22.: The registers of `as_iic`

The tables 7.23 and 7.24 explain the purpose of each control and status bit in more detail.

Bit Name	Access	Bit	Description
`Start/Continue`	W	0	Start or continue a transaction. This control bit is used to initiate a transaction from the ready state or continue a transaction for another byte after sending or checking for an acknowledge. This bit is reset after sending/receiving a byte to/from the *i2c* bus.
`Stop`	W	1	Stop a transaction. This control bit overwrites the Start/Continue bit and explicitly stops the current transaction after the next acknowledge or after the start bit. This bit is reset when the hardware is in the ready state.
`Read/Write`	W	2	Choose to write or read the next byte. This bit toggles between writing a data byte onto the bus or reading a data byte from the bus. It is sampled after checking for or sending an acknowledge. This bit is only reset after stopping a transaction. The default value is ’0’ ($\hat{=}$ write).
`Reset`	W	3	Completely reset the hardware. This bit acts just like a hard reset for the `as_iic` module. The hardware will enter the ready state again after just a few clock cycles. This bit is reset immediately.

Bit Name	Access	Bit	Description
`Data Ready`	W	4	Signal the hardware to continue. This bit operates in conjunction with the status bit ”Waiting SW”. When that status bit is set, the hardware is waiting for the software to finish setting up the data register for the next data byte or finish reading from the status register. When the software is done, it needs to set this control bit. The hardware will then continue with the transaction. This bit is reset immediately.
`Ack Mod`	W	5	Send a master acknowledge. This bit's only purpose is to tell the hardware to send an acknowledge after sending the *i2c* slave address. It is reset after sending/receiving a data byte.

Table 7.23.: Bit fields of the control register

Bit Name	Access	Bit	Description
`IIC Ready`	R	0	The module's hardware is ready. This status bit is only set when the hardware is in the ready state.
`IO Ready`	R	1	The AXI Slave Registers are safe to read/write. When this status bit is set, the hardware is currently not reading/writing from/to the data byte parts of the AXI Slave registers, meaning that IO operations are allowed and safe.
`Bus Active`	R	2	This module is active on the *i2c* bus. This status bit is always set when the `as_iic` module is actively setting the SDA signal of the *i2c* bus.

Bit Name	Access	Bit	Description
`Ack Rec`	R	3	Acknowledgement was received. This bit is set after checking for an acknowledgement bit from the slave after sending a data byte to the *i2c* bus. It should only be sampled after a write transaction as the value is only valid then. Also note that this bit can only report on the state of the acknowledgement from the previous data byte. It is reset just before checking for an acknowledge bit or when starting a new transaction.
`Stalled`	R	4	Clock stretching is detected. This bit is set every time clock stretching is detected. Note that it is accurate to within a single system clock cycle. The bit is reset immediately after the slave releases SCL.
`Waiting SW`	R	5	The hardware is waiting for the software. This bit is working in conjunction with the control bit ”Data Ready”. When this bit is set, the hardware is waiting on the control bit ”Data Ready” to be set by the software. The hardware will always wait after sending/receiving a byte, before sending/checking for an acknowledge. This allows the software to finish tasks like reading the last received data byte and writing the next data byte to be send. This bit is immediately reset after the software sets ”Data Ready”.
Unused	R	6 - 7	Unused
`Data RX`	R	8 - 15	Data received from the bus This part of the status register is used by the hardware to transfer data received from slave devices on the bus to the software.

Bit Name	Access	Bit	Description

Table 7.24.: Bit fields of the status register

Bit Name	Access	Bit	Description
`SCL_DIV`	W	0 - 23	SCL counter compare value This value is used to reset the SCL counter, which is used to generate the *i2c* bus frequency.
`Data TX`	W	24 - 31	Data to send to the *i2c* bus This part of the data register is used to transfer the bytes for the hardware to send on the bus.

Table 7.25.: Bit fields of the data register

Besides the registers, there are the clock and hardware reset signals that go into the module and the SCL and SDA signals for the *i2c* bus, which are "inout" signals, driven by the module using tristate drivers. These signals should be connected to the outside using GPIO Pins connected to the *i2c* devices, you wish to communicate with. Though not intended, an internal *i2c* bus could be configured just the same.

7.6.2.2. Generation of the SCL signal

The data width used to configure the frequency for SCL differs, depending on the hardware configuration. It is always equal to the value of the "SCL_DIV_REGISTER_WIDTH" generic, configurable before synthesis. The data is little endian.

"SCL_DIV_REGISTER_WIDTH", the modules only generic, sets the size of the counter that is used to detect when to switch the SCL signal. Therefore a wider counter allows for lower frequencies on the bus. Also: With higher system clock frequencies a wider SCL DIV counter might be necessary to achieve the desired *i2c* bus clock frequency.

The value used to configure the counter compare value in the data register (SCL_DIV), is calculated as follows:

$$RegisterValue = (SystemFrequency/(4 * DesiredBusFrequency)) - 2$$

With that, the ideal value for the "SCL_DIV_REGISTER_WIDTH" generic is:

$$\log_2(RegisterValue)$$

Where the *SystemFrequency* is the frequency of the clock driving the `as_iic` module, *DesiredBusFrequency* is the desired frequency of the SCL signal on the *i2c* bus and *RegisterValue* is the value to set the "Frequency for SCL" part of the data register to.
Note that the practical minimum value for the register value is 3. This will run the hardware as fast as possible.

A state machine in conjunction with the aforementioned counter is used to generate the SCL clock signal. Every time the counter reaches the compare value configured via the data register, it is reset and a separate modulo 4 counter is incremented. Bit 1 of this smaller counter corresponds to the state of the SCL signal. This also means that every time bit 0 of this smaller counter changes, one quarter of the bus clock period has passed. This is an important signal for many parts of the hardware, as it changes exactly between and on the edges of the SCL clock signal. Figure 7.7 shows a simplified diagram of the counters involved in generating the SCL signal.

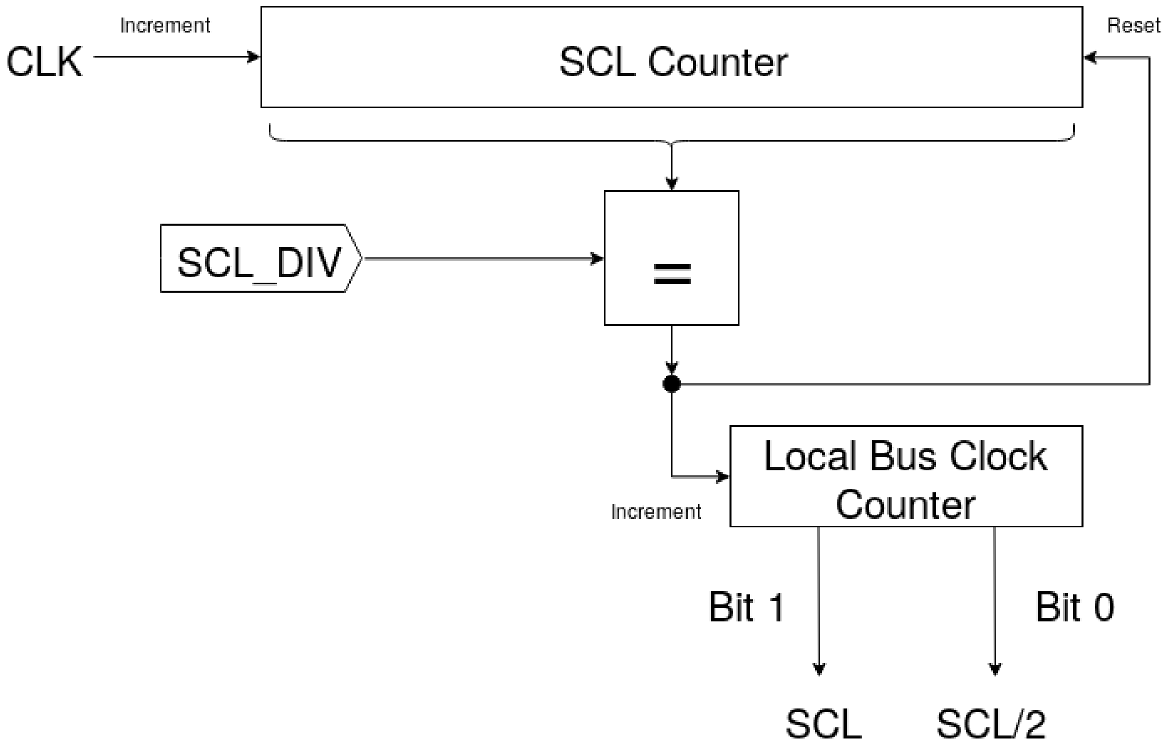

Figure 7.7.: The counters generating the SCL Signal in `as_iic`

The state machine controls the counters and sets the SCL signal according to the mod 4 counter's state. The "mod 4 counter" is also called *local bus clock* or *local bus clock counter* in hardware and in the graphic. It also sets the "stalled" status signal, whenever clock stretching is detected. This is done by monitoring the SCL bus signal and comparing it to the internal SCL signal (*lbclk*). If the signals differ from each other, another device on the *i2c* bus is interfering with the clock signal ($\hat{=}$ "clock stretching").

7.6.2.3. The SDA state machine

This second larger state machine is used to control the SDA signal, manage the communication with the software via the AXI-Bus and manage the SCL state machine.
The states of this state machine can be grouped to correspond to different sections of the *i2c* protocol, as seen in figure 7.8.

The state machine oftentimes uses the *i2c* bus signals SDA and SCL as parameters. When it does that, these signals are not the internal signals, but are read directly from

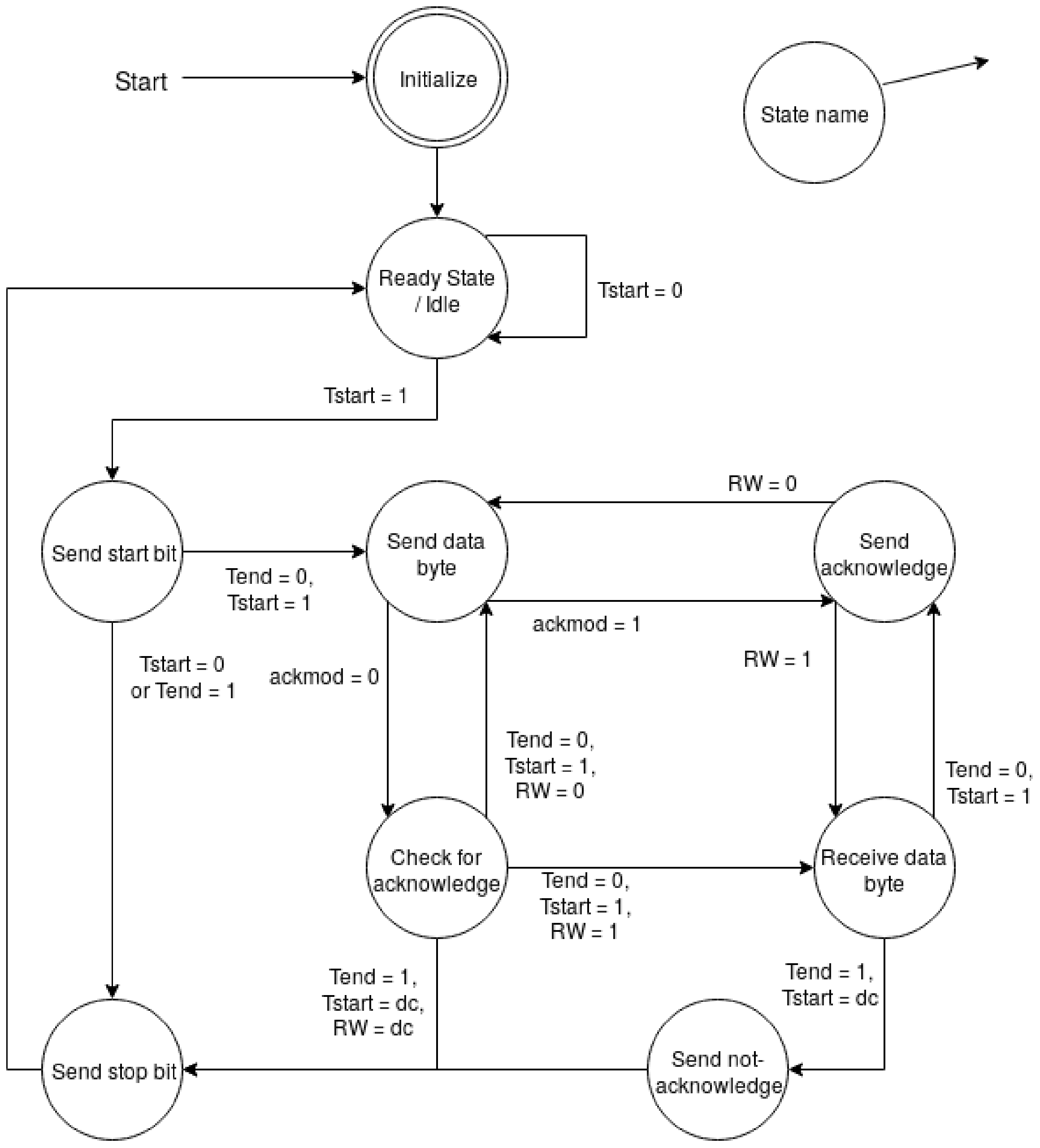

Figure 7.8.: Visualization of the SDA state machine in `as_iic`

the bus, to make sure that clock stretching is always recognized.
When setting or sampling the SDA signal while sending or receiving a data byte, bit 0 from the *local bus clock counter* (*lbclk_half*) is used as the trigger. When this bit of the counter changes, exactly one quarter of the SCL clock period has passed, meaning that this is exactly in between two SCL clock edges. This guards against possible timing violations.

7.6.2.4. The "Master Acknowledge"

A non-standard feature supported by this *i2c* master implementation is the *Master-Acknowledge*. After the slave address has been sent to the bus, the master is able to send an acknowledge bit by itself. This behaviour is controlled by the software through the `ack_mod` control bit. Some driver functions, like `set_regpointer`, use this functionality and some functions take an additional parameter, the `modifier` byte, which can enable this and other functionality.

7.6.2.5. How to connect a hardware module to the *i2c* bus

As mentioned before the *i2c* bus consists of just two signals/wires: SCL and SDA. To connect a new slave device to the bus, these two wires of the `as_iic` master need to be connected to the appropriate wires of the slave. The SCL wires are connected together and the SDA wires are connected together.
The *i2c* bus requires that each signal has a pull-up resistor connected to it. This requires a resistor of between 1 kilo ohm and 10 kilo ohm to be connected to the signal and the supply voltage (usually 3.3 Volts) for both signals.
Possible problems with the *i2c* bus include:

- Multiple pull-up resistors present per signal
- Capacitance between the signals and ground is too high
- Interference from other signals

For more in-depth knowledge on the design of the hardware, consider looking through the VHDL source files of `as_iic` available in `"modules/as_iic/hardware/"`.

7.6.2.6. Waveform examples

This section will further explain the *i2c* protocol, using some waveform examples.
Figure 7.9 shows some signals of a simulated `as_iic` during a write operation.
The first operation shown is the start bit, SDA going low while SCL is high followed by SCL going low while SDA is still low. Following that, SDA may change while SCL is low and has to be valid when SCL goes high.
Furthermore some `as_iic` specific relations can be demonstrated using this waveform. The relationship between the system clock `clk`, the local bus clock counter, `lbclk` ($\hat{=}$ bit 1) and `lbclk_half` ($\hat{=}$ bit 0), SCL and SDA where SCL is equal to `lbclk` but slightly delayed and SDA changes slightly after `lbclk_half` goes high.
Figure 7.10 shows a read transaction from the same simulated module.
Finally figure 7.11 shows a complete read transaction and write transaction captured using a logic analyzer using real hardware. In this figure the start bits are marked by a

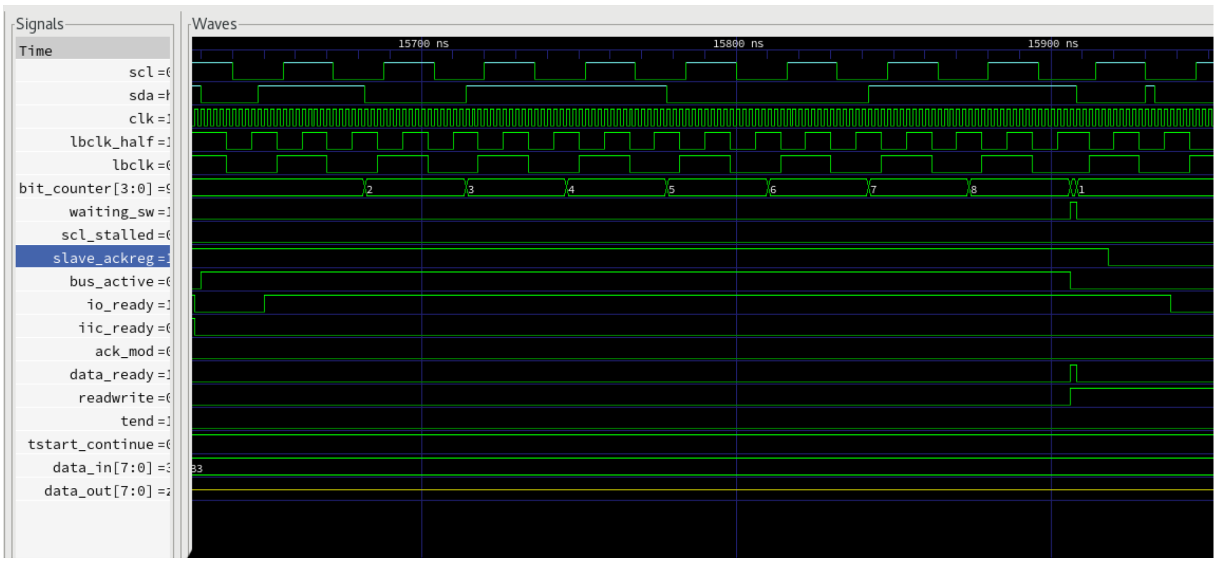

Figure 7.9.: *i2c* read transaction as a waveform from a simulated `as_iic` module

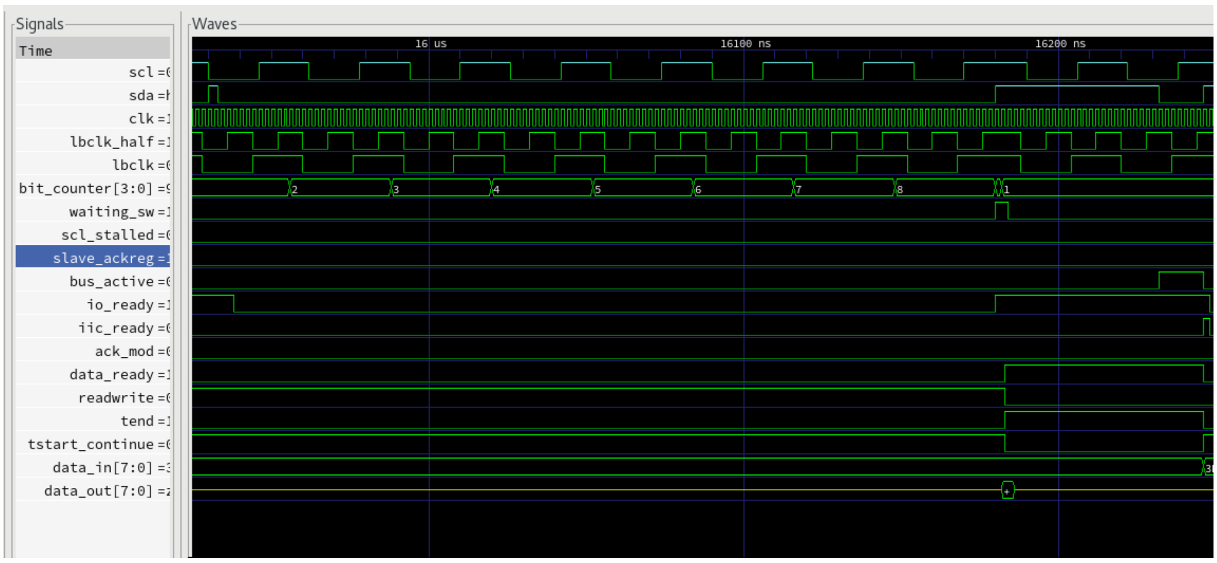

Figure 7.10.: *i2c* write transaction as a waveform from a simulated `as_iic` module

green circle and the stop bits by a red circle. Every transferred bit is marked by a small arrow on the edge of SCL going high, with the clock cycle for the acknowledge lacking this arrow.

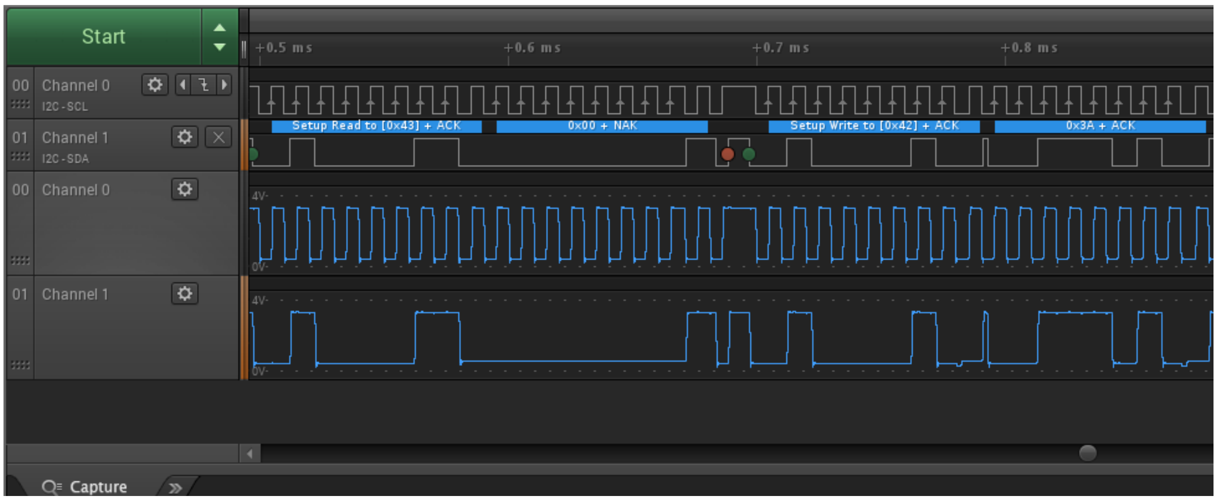

Figure 7.11.: Waveform captured using a logic analyzer from the `as_iic` module

7.6.3. The Software Driver

The driver's low level functions are built to act as a modular system. A few low level functions can be put together to a sequence, which can act as any possible *i2c* transaction the hardware supports. This means that less knowledge of this particular hard- and software implementation is required to expand the drivers functionality. All of the lower level functions reference "transactions". This refers to all the actions between and including the start bit and the stop bit, meaning a transaction can contain an arbitrary count of data bytes.

The higher level functions can be used as is, after initializing the module properly. They include single byte transactions, multi-byte transactions and transactions which refer to "registers" or "reg(s)". This refers to registers of the *i2c* slaves on the bus.

Table 7.26 provides an overview of the transactions available in the driver.

Function Name	**Transaction Type**
as_iic_write_byte	Single Byte Write
as_iic_get_byte	Single Byte Read
as_iic_write_bytes	Multi-Byte Write
as_iic_read_bytes	Multi-Byte Read
as_iic_write_reg	Set an *i2c* Device's Slave Register
as_iic_read_reg	Read an *i2c* Device's Slave Register
as_iic_read_regs	Read successive Slave Registers
as_iic_set_regpointer	Set special Slave Register (Master ACK)

Table 7.26.: Listing of the high-level functions available for `as_iic`

For more information on the high level functions and the lower level functions not mentioned here, see the *ASTERICS* Doxygen driver documentation.

7.6.4. Quirks of `as_iic`

7.6.4.1. Hard Wait Mechanism

The driver has a hard-coded wait mechanism that waits for about 50 us after every transaction to give the slave enough time to recognize the end and start of two sequential transactions.

7.6.4.2. SCL Frequency Configuration

The configuration for the SCL frequency is immediately applied in the hardware. The valid frequencies range from 10kHz to 1MHz. Note though, that the entire range is not always supported by the specific hardware configuration. This means, that the frequency can be set to a value that the `as_iic` hardware can not handle. The minimum value for the SCL configuration register is 3. The maximum is dependent on your hardware configuration of *ASTERICS* . Note that the frequency configuration is NOT reset when calling the reset-function `as_iic_reset_hw_state()` and is only reset by the function `as_iic_reset()`.

7.6.4.3. Pull-up Resistors for the *i2c* Bus

The `as_iic` module does not configure internal pull-up resistors for the *i2c* signals, as the internally provided current is usually too weak. Therefore external pull-up resistors have to be provided by the user in order to use the `as_iic` module. The section 7.6.2.5 briefly covers how to connect devices to the *i2c* bus and how to connect the required pull-up resistors.

7.7. The *VEARS* module

by Michael Schaeferling

This section describes the `VEARS` module in detail. `VEARS` stands for ”Visualization for Embedded Augmented Reality Systems”. It is developed to display an image on a monitor and enrich this image by a graphical overlay, e.g. to mark particular image regions or to display other information (like text) on top of the image (without manipulating the original image stored in the main memory). Although `VEARS` is part of the *ASTERICS* framework, unlike many other *ASTERICS* modules, it is a self-contained IP core. The `VEARS` module was initially developed as a project work by several students of the University of Applied Sciences, Augsburg. Since then, it is maintained by the EES workgroup.

7.7.1. Brief description

The `VEARS` module is a stand-alone IP core which can be integrated into a system on chip, also without the need of an *ASTERICS* image processing chain. The image and the overlay are stored in the systems main memory where `VEARS` fetches them via AXI Master Burst accesses. The image to display is to be provided by the user in a specified format, which may be grayscale or color. In grayscale mode, `VEARS` uses 8 bits per pixel, while in color mode 32 bits per pixel (8 bits for red, green and blue each with 8 bits padding) are used. To simplify overlay manipulation, `VEARS` provides several functions for this task, e.g. for drawing lines, circles, rectangles, etc. and also to draw text. The overlay is also stored in main memory, using a space and memory bandwitdh saving 2 bit per pixel data format.

7.7.2. The Hardware

7.7.2.1. Configuration Options

`VEARS` supports a selection of most common video formats (and may be extended for other desired formats in the future). Video and color mode selection is set at synthesis time via generics as several fixed hardware structures, such as line buffers and clock generators depend on the video format and timing. Also the desired video output method, such as VGA, HDMI or interfacing to external video encoder chips is set at synthesis time. The following parameters are set by generics (which can also be accessed in the Vivado blockdesign GUI), as described in the subsequent paragraphs.

- **Video Group** and the respective **Video Mode** (see table 7.27)
- **Color Mode**
- **VGA output enable** and **VGA TFT output enable** along with**VGA Color width**
- **HDMI output enable**
- **Chrontel CH7301 output enable**
- **AXI Clock Frequency**

The following **Video Group** (1=CEA, 2=DMT) and **Video Mode** combinations are currently supported:

Video Group	**Video Mode**	Video Format / Timing	Pixel Frequency
1	4	1280x720 @60Hz/45kHz	74.250 MHz
1	32	1920x1080 @24Hz/26.8kHz	74.250 MHz
1	33	1920x1080 @25Hz/27.9kHz	74.250 MHz
1	34	1920x1080 @30Hz/33.5kHz	74.250 MHz
2	4	640x480 @60Hz/31.5kHz	25.175 MHz
2	8	800x600 @56Hz/35.2kHz	36 MHz
2	10	800x600 @72Hz/48.1kHz	50 MHz
2	16	1024x768 @60Hz/48.4kHz	65 MHz
2	35	1280x1024 @60Hz/64kHz	108 MHz

Table 7.27.: `VEARS` - Supported Video Modes

The **Color Mode** for the image can be '0' (8-bit grayscale) or '1' (24-bit color). In grayscale mode, image pixels are stored as consecutive 8-bit values in memory. In color mode, each pixel occupies 32 bits in memory where 8 bits are used for red, green and blue channels (resulting in 24 bit RGB) and 8 bits are used for padding. Note that color mode does only affect the image and not the overlay (which is a fixed 2 bits per pixel format).

The **VGA, HDMI and CH7301 output enables** should be set accordingly to the desired output methods. **VGA TFT output enable** is an extension to the VGA output, needed by some digital displays.

AXI Clock Frequency must be set to the actual system bus frequency as it is used to calculate internal parameters for generating the video clock (the systems bus clock is used as a clock source). In Xilinx Vivado block-designs, this value should be updated automatically.

7.7.2.2. Interrupts

`VEARS` provides interrupt output signals *intr_frame* and *intr_line*, e.g. in order to synchronize software. These signals are active high at the beginning of the video sync time (V-Sync for *intr_frame* and H-Sync for *intr_line*) for one AXI-slave clock cycle. The interrupt signals can be controlled by enable bits of the control register (see Table 7.29).

7.7.2.3. Considerations to Memory Bandwidth

When selecting the required **Video Group/Video Mode** in combination with the **Color Mode**, it should be considered that there is enough memory bandwidth available for image data transfer. `VEARS` pre-fetches image and overlay data for the next line on-the-fly while outputting the recent line to the monitor. Thus, image and overlay data must be fetched into the internal line-buffer within the recent lines time-to-draw. To meet this requirement, the AXIs clock speed must be set high enough so that the bus is able to transport at least image and overlay data (if `VEARS` is the only bus master). As a rule of thumb, in color mode one should budget the system bus to be occupied by at least 1.25x the recent video modes pixel frequency. In grayscale mode, only a quarter of this bandwidth is needed as for image data only 8 bits instead of 32 bits (for color) have to be transferred per image pixel. The overlay has very little impact on memory bandwidth usage as it uses a space and bandwidth saving 2 bit per pixel data format, but may also be taken into account for bandwidth considerations.

7.7.2.4. Pitfall: SoC software re-upload

The `VEARS` module continuously fetches data when it is enabled. In Xilinx Zynq environments it was observed that this can cause a problem when the system software is re-uploaded. During various initialization steps which are automatically performed on software upload and start (so called ”ps7init”) the Zynq-system is getting prepared for operation (several Zynq PS register values for clocking etc. are set), but if `VEARS` is still running during that time (as it may not be disabled before re-uploading the software), this initialization phase is likely to fail (due to pending bus transactions caused by `VEARS`). The only recovery option is to power-cycle the system.

Thus, on Zynq-based (and probably other) systems, `VEARS` must be disabled before re-uploading an running software on the system once it was enabled before!

7.7.2.5. Register Space

The `VEARS` module is configured by 32 bit wide slave registers, connected to the AXI-Slave bus. Table 7.28 gives an overview on the registers.

Register Name	Access	Offset	Description
`Control Register`	W	0	Control register for the hardware: Various bits are used to control the module. See table 7.29 for a detailed description.
`Status Register`	R	1	Status register for the hardware: This register delivers various information on the capabilities of the module. See table 7.30 for a detailed description.
`Image Base-Address`	W	2	The image base-address: used to fetch image data from memory.
`Overlay Base-Address`	W	3	The overlay base-address: used to fetch overlay data from memory.
`Overlay Color 1`	W	4	Overlay Color 1: 24 bit RGB palette value of overlay color 1.
`Overlay Color 2`	W	5	Overlay Color 2: 24 bit RGB palette value of overlay color 2.
`Overlay Color 3`	W	6	Overlay Color 3: 24 bit RGB palette value of overlay color 3.

Table 7.28.: The registers of `VEARS`

Tables 7.29 and 7.30 explain the purpose of each control and status bit in more detail.

Bit Name	Access	Bit	Description
`Reset`	W	0	Reset the VEARS module. This control bit can be used to reset the `VEARS` module.

Bit Name	Access	Bit	Description
`Enable`	W	1	Set the `VEARS` instance into operation. This control bit is used to activate the `VEARS` module. When activated, the `VEARS` module will grab image data from memory. For this, an appropriate image base address must be supplied (via the according register) before enabling `VEARS`. Note: the `VEARS` module will generate video data on the monitor output ports (a vertical bit pattern) even if it's not enabled.
`Overlay Enable`	W	2	Enable the overlay. When the `VEARS` module is in operational mode (bit "Enable" is set), the overlay can be enabled or disabled with this bit separately. An appropriate overlay base address must be supplied (via the according register) before enabling overlay output.
`Frame Interrupt Enable`	W	6	Enable frame interrupt. Each time a new frame starts (at the start of V-Sync) an interrupt signal is generated on the *intr_frame* output.
`Line Interrupt Enable`	W	7	Enable line interrupt. Each time a new line starts (at the start of H-Sync) an interrupt signal is generated on the *intr_line* output.

Table 7.29.: Bit fields of the control register

Bit Name	Access	Bit	Description
`Video Group`	R	[7:0]	Video Group: These bits give information on the video group supported by this `VEARS` instance. **Video Group** and **Video Mode** can be used to determine the video output format.

Bit Name	Access	Bit	Description
`Video Mode`	R	[15:8]	Video Mode: These bits give information on the video mode supported by this `VEARS` instance. **Video Group** and **Video Mode** can be used to determine the video output format.
`Color Mode`	R	16	Color Mode: This bit gives information on the color mode supported by this `VEARS` instance: '0': grayscale '1': color

Table 7.30.: Bit fields of the status register

7.7.3. The Software Driver

The driver functions can be split to two categories: the hardware interfacing functions and functions for overlay manipulation.

Hardware interfacing functions are used to generally control the module, such as to enable or disable the module at all or to set memory adresses for image and overlay data.

Overlay manipulation functions can be used to erase the whole overlay, to draw lines, circles or rectangles or even to draw text to the overlay.

For a detailed overview on the software driver functions, see the *ASTERICS* Doxygen driver documentation.

8. Complex Modules

`- This chapter is currently under construction -`

9. Systems

This section describes systems provided along with the *ASTERICS* distribution, e.g. for demonstration of the *ASTERICS* frameworks abilities.

9.1. as_refdesign_zynq

by Michael Schäferling, Philip Manke

This system demonstrates how a minimal *ASTERICS* system may be assembled and prepared to be runnable out-of-the-box on evaluation boards which are based on the Xilinx Zynq platform. Currently only the Zybo-Board is fully supported. The system generation is tested with Vivado 2017.2, 2018.3 and 2019.1. Note that necessary cable drivers and board files need to be installed.
The image processing chain is controlled by bare-metal software running on an ARM core and consists of *ASTERICS* hardware modules for image capturing, basic image operations and writing the resulting image to system memory. The system also includes an *ASTERICS* module for visualization (VEARS) which allows to observe the output image stream on an attached screen connected via HDMI or VGA.

This demo system implements an image difference calculation on the FPGA.

The system architecture is depicted in figure 9.1. The OV7670 camera is directly connected to the programmable logic and interfaces with an *ASTERICS* module `as_sensor_ov7670`. The physical connection is done via a adapter board or fly-wire connections. This is detailed in the `doc` directory of the demo system. This module converts the camera data stream into a standardized `as_stream` inteface that the other modules understand. The data stream is duplicated by the `as_stream_splitter` module. Each frame is written to RAM by `as_memwriter0` from where it is read back when the next frame arrives by the `as_memreader0`. This has the effect of a delay of one frame for this data stream. The delayed (previous) and duplicated (current) frame are then synchronized by the `as_stream_sync` module and each pixel pair is subratcted from each other by the `as_pixel_diff` module. The resulting image is then stored in RAM for visualization or further processing. The `as_collect` and `as_disperse` modules are used to pack and unpack the eight bit pixel data into 32 bit words for more efficient memory access. The software for this system is only used to initialze and control the camera and memory access modules and the VEARS core.

This system serves as an example of the capabilities of *ASTERICS* and may be used as a starting point and style guide for building your own image processing system using *ASTERICS* . When implemented on the ZyboBoard, switch zero is used to switch between showing the buffered original camera image and the difference image on the screen using the VEARS IP-Core.

For further details regarding buiding and testing this system, please refer to the included README file and sections 2.2.1 and 6.2.1.

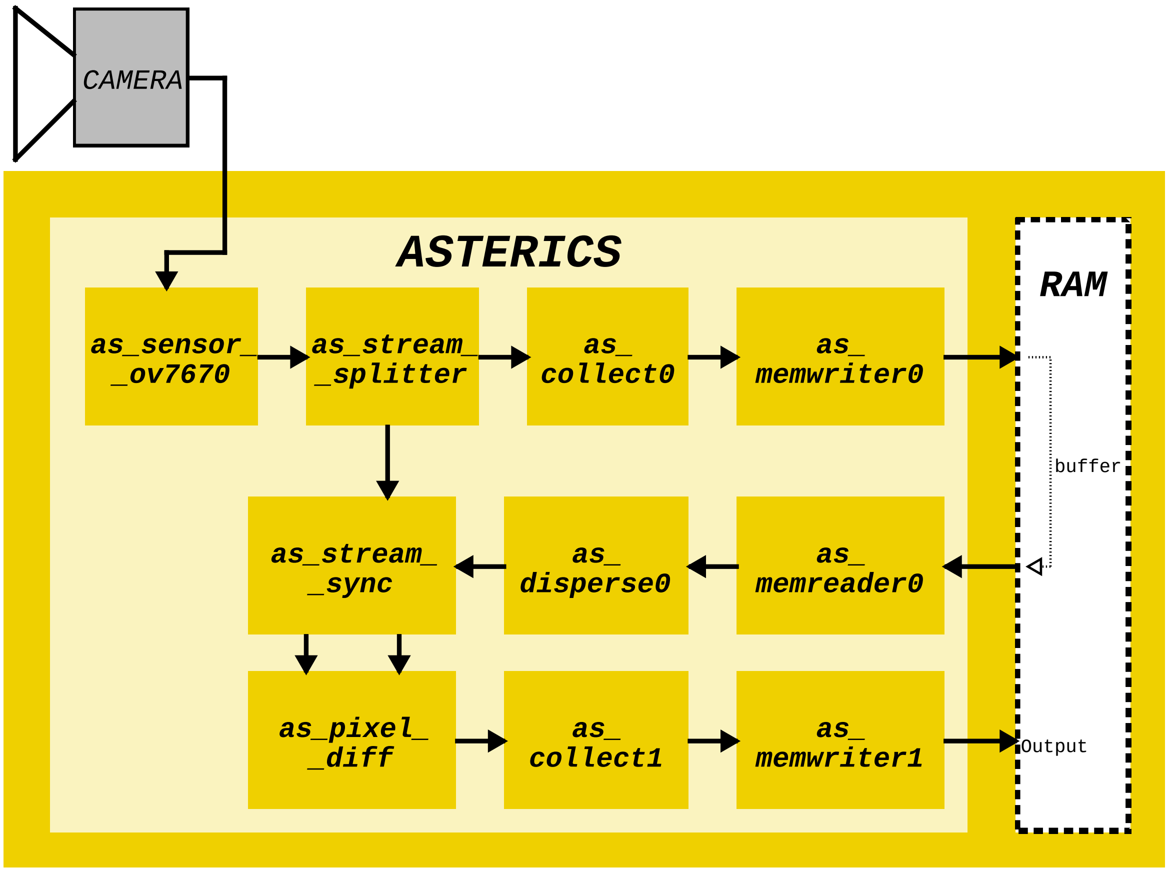

Figure 9.1.: Dataflow diagram of the example *ASTERICS* system, depicting the included modules.