padrick Documentation

Release 0.3.6

Manuel Eggimann

Dec 14, 2022

CONTENTS

Warning: Padrick is still in it's very early stage of development and might still experience refactoring and breaking changes in the future. Also, while we do our best to test and verify the padrick generated IPs and have successfully used it in actual silicon tapeouts we do not provide any guarantees for the correctness of the generated RTL. You should always verify the generated IP yourself.

Modern SoCs contain a large number of peripherals and functional blocks that need to communicate with the outside world. Most of the time, the number of IO pins required to map every IO facing port of the whole SoC to dedicated IO pads is infeasible due to limited available area dedicated for IO pads. Instead, most modern SoCs use multiplexing logic to use a single IO pad for several different functionalities according to some user-programmable configuration register. While some SoCs (e.g. Nordic nRF52 series of BLE chips) allow mapping of every module port (e.g. SPI MOSI signal) to every available IO pad in a full crossbar fashion, traditionally the PULP chips taped out so far only used the approach to assign 2-3 Module ports to one dedicated IO pad. I.e. the user can chose to use e.g. IO Pad 43 as either GPIO4 or I2C_SCK but the SPI_MOSI signal for example cannot be routed to IO Pad 43. While the full-crossbar like approach can have its demerits (e.g. routing can become trickier if very fast IO signals are involfed), the full-crossbar routing approach makes the resulting SoC way more flexible for dynamic adaptation to the workload and simplifies a lot the PCB design process and repurposing of existing PCBs by means of rerouting the IO signals.

Since the padframe is always a custom tailored component for one particular SoC, a lot of time has been spent in the past writing this padframe multiplexing and routing logic. This design process is very labouruos and extremely error prone which is where Padrick enters the stage;

Padrick is a command line tool written in Pyhton3 that aims to solve the problem of painstakingly writing the multiplexing logic and interconnection logic for padframes by hand. Not only does it allow full-crossbar like routing of every periphery port to every IO pad it also generates C-drivers and Documentation to interact with the auto-generated IP. Every aspect of the generated pad multiplexing IP is fully customizable and technology agnostic. The degree of multiplexing capability can be customized from anything between static pad assignments to full any-to-any crossbar routing to also support more traditional pad multiplexing schemes.

How does it work? The user provides a YAML description of the desired Padrame configuration. The configuration file's syntax is tailored in a way such that large portions of it can be copy pasted or even entirely reused from one SoC to the other while only minor modifications of e.g. the involved IO pads need to be performed when porting to different technologies or when reusing IO peripherals. After parsing and internaly validating the configuration file with a couple of sanity checks, Padrick generates several SystemVerilog modules that instantiate the desired IO Pads, implements the IO Multiplexing logic and exposes the SoC facing IO signal and an auto-generated configuration register file that can be accessed through a dedicated configuration interface (a lightweight protocol which can be easily conveted to AXI-lite or APB) to be attached to the SoC configuration bus for at-runtime control over the IO Pads.

CHAPTER

ONE

CONTENTS

1.1 Padrick - A flexible Pad Multiplexer Generator for SoCs

Warning: Padrick is still in it's very early stage of development and might still experience refactoring and breaking changes in the future. Also, while we do our best to test and verify the padrick generated IPs and have successfully used it in actual silicon tapeouts we do not provide any guarantees for the correctness of the generated RTL. You should always verify the generated IP yourself.

Modern SoCs contain a large number of peripherals and functional blocks that need to communicate with the outside world. Most of the time, the number of IO pins required to map every IO facing port of the whole SoC to dedicated IO pads is infeasible due to limited available area dedicated for IO pads. Instead, most modern SoCs use multiplexing logic to use a single IO pad for several different functionalities according to some user-programmable configuration register. While some SoCs (e.g. Nordic nRF52 series of BLE chips) allow mapping of every module port (e.g. SPI MOSI signal) to every available IO pad in a full crossbar fashion, traditionally the PULP chips taped out so far only used the approach to assign 2-3 Module ports to one dedicated IO pad. I.e. the user can chose to use e.g. IO Pad 43 as either GPIO4 or I2C_SCK but the SPI_MOSI signal for example cannot be routed to IO Pad 43. While the full-crossbar like approach can have its demerits (e.g. routing can become trickier if very fast IO signals are involfed), the full-crossbar routing approach makes the resulting SoC way more flexible for dynamic adaptation to the workload and simplifies a lot the PCB design process and repurposing of existing PCBs by means of rerouting the IO signals.

Since the padframe is always a custom tailored component for one particular SoC, a lot of time has been spent in the past writing this padframe multiplexing and routing logic. This design process is very labouruos and extremely error prone which is where Padrick enters the stage;

Padrick is a command line tool written in Pyhton3 that aims to solve the problem of painstakingly writing the multiplexing logic and interconnection logic for padframes by hand. Not only does it allow full-crossbar like routing of every periphery port to every IO pad it also generates C-drivers and Documentation to interact with the auto-generated IP. Every aspect of the generated pad multiplexing IP is fully customizable and technology agnostic. The degree of multiplexing capability can be customized from anything between static pad assignments to full any-to-any crossbar routing to also support more traditional pad multiplexing schemes.

How does it work? The user provides a YAML description of the desired Padrame configuration. The configuration file's syntax is tailored in a way such that large portions of it can be copy pasted or even entirely reused from one SoC to the other while only minor modifications of e.g. the involved IO pads need to be performed when porting to different technologies or when reusing IO peripherals. After parsing and internaly validating the configuration file with a couple of sanity checks, Padrick generates several SystemVerilog modules that instantiate the desired IO Pads, implements the IO Multiplexing logic and exposes the SoC facing IO signal and an auto-generated configuration register file that can be accessed through a dedicated configuration interface (a lightweight protocol which can be easily conveted to AXI-lite or APB) to be attached to the SoC configuration bus for at-runtime control over the IO Pads.

1.2 Getting Started

1.2.1 Installing Padrick

Padrick is a CLI tool that you invoke on a configuration file that describes the structure of your SoCs pad multiplexing and IO mapping scheme. The very first step of ussing padrick is thus installing it. There are two ways to install padrick on your system;

1.2.1.1 Downloading a Binary Distribution/Using Padrick without Python

The preferred installation method if you don't modify padrick or need the most bleeding edge version of it is to just use a self-sufficient binary. This is especially usefull if your development environment does not provide a reccent Python 3 installation or you are unable to install any additional python dependencies. The binary appimage distribution of Padrick wraps its own python interpreter in a Rust executable to interpret the padrick python source code embedded within the binary itself (this is enabled by a project called Pyoxidizer). Any Linux distribution with glibc version 2.14 or newer should be able to run the Appimage binary. This includes but is not limited to the following or newer Linux Distributions:

- Debian 8
- Fedora 16
- OpenSUSE 12.1
- RHEL/CentOS 7
- Ubuntu 12.04

You can find the latest binary x86 release on the github release page.

Use the following snippet to download the appimage in your current path:

```
curl https://api.github.com/repos/pulp-platform/padrick/releases/latest \
| grep "Padrick-x86_64.AppImage" \
| cut -d : -f 2,3 \
| tr -d \" \
| wget -qi -
mv Padrick-x86_64.AppImage padrick
chmod a+x padrick
```

Now you can directly start using the downloaded binary. E.g. use this command to show the built-in help:

```
./padrick --help
```

1.2.1.2 Installing Padrick as a Python Package

If you have python3.6 or newer available on your system, you can directly install padrick using `pip`:

```
pip install git+ssh://git@github.com:pulp-platform/padrick.git
```

Or if you prefer https over ssh:

```
pip install git+https://github.com/pulp-platform/padrick.git
```

If you plan to modify or frequently update padrick you might want to install it with the pip editable flag so changes to the source code of padrick take effect immediately to all Python environments were you installed padrick:

```
git clone https://github.com/pulp-platform/padrick.git
pip install -e ./padrick
```

These approaches will install all the required python dependencies automatically and make the command line tool `padrick` available for your shell.

1.2.2 Writing a Padframe Configuration File

The next step after installing Padrick is to write a configuration file for your padframe. The configuration file captures all information about the padframe required for your SoC, from IO cell specifications, IO peripheral signal declaration to multiplexing strategy. The config file is written in YAML, a powerful, human readable markup language. The following listing shows you a minimal padframe configuration file to generate a simple padframe for an SoC with 4 pads an SPI and a UART peripheral where each signals of the UART or SPI peripheral can be routed to anyone of the four available pads.

```
manifest_version: 2
name: my_padframe
pad_domains:
  - name: my_domain
    pad_types:
      - name: iocell_xy
        template: |
          IOLIB_IOCELL_XY ${instance_name} (
            .PAD(${conn["pad"]}),
            .I(${conn["chip2pad"]}),
            .O(${conn["pad2chip"]}),
            .OUT_EN(${conn["tx_en"]})
          );
        pad_signals:
          - name: pad
            size: 1
            kind: pad
          - name: chip2pad
            size: 1
            kind: input
            conn_type: dynamic
            default_reset_value: 0
            default_static_value: 1'b0
          - name: pad2chip
            description: "The signal that connects to the pads RX buffer"
            size: 1
            kind: output
            conn_type: dynamic
          - name: tx_en
            description: "Active high RX driver enable "
            size: 1
            kind: input
            conn_type: dynamic
            # by default, the output driver is disabled
            default_reset_value: 1
            default_static_value: 1'b1
    pad_list:
```

(continues on next page)

(continued from previous page)

```
    - name: iopad_{i}
      multiple: 4
      pad_type: iocell_xy
  port_groups:
    - name: SPIM
      output_defaults: 1'b0
      ports:
        - name: miso
          connections:
            miso: pad2chip
            tx_e: 1'b0
        - name: mosi
          connections:
            chip2pad: mosi
            tx_en: 1'b1
        - name: sck
          connections:
            chip2pad: sck
            tx_en: 1'b1
        - name: cs
          connections:
            chip2pad: cs
            tx_en: 1'b1
    - name: UART
      output_defaults: 1'b0
      ports:
        - name: rx
          connections:
            uart_rx: pad2chip
            tx_en: 1'b0
        - name: tx
          connections:
            chip2pad: uart_tx
            tx_en: 1'b1
```

The different keys and settings in this example might seem confusing at the moment, but they are all explained in detail in chapter *Padframe Configuration File*. For the purpose of this introductionary tutorial, just copy the content of the example to a new file and give it the name my_padrame_config.yaml

1.2.3 Validating the Configuration File

Now that we wrote our first configuration file, it is time to validate it. Padrick contains extensive validation checks. Not only does it make sure that the configuration file is properly formated and contains all required keys with corresponding value of the right type, it also runs a number of sanity checks on your configuration to detect semantic mistakes e.g. IO signals without corresponding pads or naming conflicts. While padrick always validates your config file before rendering any output there is a dedicated CLI command to run validation only:

```
padrick validate my_padframe_config.yaml
```

If you copied the example above you will see a user friendly error message pointing out a typo in your config file. On line 46 there is a type: The connection entry should be *tx_en: 1'b0* instead of *tx_e: 1'b0*. Correct the mistake and validate the config file once again. Now you should not encounter any errors.

1.2.4 Generating the RTL for the Padframe IP

Now that we validated the syntactic (and to some degree semantic) correctness of our configuration file it is time to generate the padframe. To do so, type the following command:

```
padrick generate rtl my_padframe_config.yaml -o my_padframe_ip
```

This will generate a new folder called *my_padframe_ip* in your current directory and renders the complete padframe IP. The generated IP instantiates our IO pads using our specified IO cells, generated the multiplexing logic to route our IO peripheral signals (SPI and UART) to one of those pads and instantiates a register file to configure the connectivity and the configuration of the IO pads through some configuration interface.

A closer inspection of the folder content reveals the following folder structure:

```
my_padframe_ip
├── Bender.yml
├── ips_list.yml
├── src
│   ├── my_padframe_my_domain_config_reg_pkg.sv
│   ├── my_padframe_my_domain_config_reg_top.sv
│   ├── my_padframe_my_domain_muxer.sv
│   ├── my_padframe_my_domain_pads.sv
│   ├── my_padframe_my_domain_regs.hjson
│   ├── my_padframe_my_domain.sv
│   ├── my_padframe.sv
│   ├── pkg_internal_my_padframe_my_domain.sv
│   └── pkg_my_padframe.sv
└── src_files.yml
```

At the top-level, there are some IP manifest files that simplify the integration of our IP in an SoC using an IP dependency management tool.

Hint: *Bender.yml* is used for the more modern PULP IP management tool Bender while *src_files.yml* and *ips_list.yml* are required for usage with the legacy pulp IP tool IPApproX.

The *src* directory contains all the generated SystemVerilog source files where *my_padframe.sv* contains the toplevel module. Let's have a look at the interface of this module:

```
module my_padframe
  import pkg_my_padframe::*;
#(
  parameter int unsigned   AW = 32,
  parameter int unsigned   DW = 32,
  parameter type req_t = logic, // reg_interface request type
  parameter type resp_t = logic, // reg_interface response type
  parameter logic [DW-1:0] DecodeErrRespData = 32'hdeadda7a
)(
  input logic                                clk_i,
  input logic                                rst_ni,
```

(continues on next page)

(continued from previous page)

```
  output port_signals_pad2soc_t                 port_signals_pad2soc,
  input port_signals_soc2pad_t                  port_signals_soc2pad,
  // Landing Pads
  inout wire logic                              pad_my_domain_iopad_0_pad,
  inout wire logic                              pad_my_domain_iopad_1_pad,
  inout wire logic                              pad_my_domain_iopad_2_pad,
  inout wire logic                              pad_my_domain_iopad_3_pad,
  // Config Interface
  input req_t                                   config_req_i,
  output resp_t                                 config_rsp_o
  );

  ...
```

Apart from a clock and reset signal, the module exposes the IO peripheral signals for UART and SPI peripheral (*port_signals_pad2soc`and `port_signals_soc2pad*, the inout wire signals for the instantiated IO cell landing pad signals (which you will probably want to route to the toplevel interface of your chip) and a configuration interface so the SoC can change the padframe configuration at runtime.

Note: At the moment, the only supported configuration interface protocol is the lightweight Register Interface Protocol. The linked github repository contains easy to use protocol converters to various other protocols like AXI, AXI-lite or APB. In the near future, Padricks *generate rtl* will command will provide a flag to directly embed the required protocol converters within the generated module exposing the protocol of your liking to the toplevel.

1.2.5 FuseSoC Support

Since version v0.3.5 Padrick has built-in support for FuseSoC. That is, it generates FuseSoC core files as part of the RTL generation process and the CLI containts a dedicated subcommand for Padrick to behave as a FuseSoC generator. In order to integrate Padrick into your flow you can copy the generator core file and the invocation script from the *fuseSoC_generator* directory in the main repository into your project.

Like any FuseSoC generator, you supply *parameters* to padrick when you call the generator in your *generate* sections. Here is an example of a small core file to generate a padframe:

```
CAPI=2:

name: "padrick:ip:padframe"
description: "My SoC's padframe"


filesets:
  padframe_deps:
    depend:
      - pulp-platform.org::common_cells:^1.21.0
      - pulp-platform.org::register_interface:^0.3.1
      - pulp-platform.org:utils:padrick

generate:
  padframe_rtl:
    generator: padrick
```

(continues on next page)

(continued from previous page)

```
    parameters:
      padrick_cmd: padrick
      generate_steps:
        - kind: rtl
      padframe_manifest: padframe.yaml

targets:
  default:
    filesets:
      - padframe_deps
    generate:
      - padframe_rtl
```

At the very beginning of the core file we register a couple of cores as dependencies since the auto-generated padframe makes use of some of their modules internally. They are:

- common_cells
- register_interface

As you can see, the *parameters* sections contains three essential key-value pairs:

padrick_cmd
: This parameter tells the small *padrick_generator.py* script how to find and invoke padrick. The command you mention here will first be looked up in your PATH and if it cannot be found there, it will try to find an executable relative to this core file to inoke. In other words you can either point to a downloaded padrick binary or just rely on the specified command being in your PATH (e.g. if you installed padrick into your python environment).

generate_steps
: Here you specify what padrick should generate for you as a list of step entries. The following *kind* of generate steps are currently supported:

  RTL Generation Step:

```
- kind: rtl
```

This entry tells padrick to generate all the RTL output files, as if you were the invoke the *generate rtl* subcommand of Padrick's CLI.

  Custom Template Rendering Step:

```
- kind: custom
  template_file: my_custom_mako_template_file.sv.mako
  output_filename: my_pad_list.csv
```

This generate step invokes padrick's custom template render command with the provided template file (relative to the current core file) and the desired output Path (generated relative to the FuseSoC managed build directory for generators). In contrast to the RTL generate step, you can register multiple custom rendering commands with different template files and targets.

padframe_manifest
: In this required parameter you tell padrick where to find the padframe configuration YAML file. The path is once again relative to the location of the calling core file I.e. in the example above it expects to find the file *padframe.yml* right next to the core file itself. The output of Padrick is generated in a build directory auto-created by FuseSoC for every Generator and automatically registered in your build dependencies. Checkout FuseSoC's Generator documentation for more information.

1.2.6 Next Steps

You now should be a bit more familiar what Padrick is, what it can do for you and how to run it. In order to actually use it, you need to get familiar with the details of the configuration file syntax and the available CLI commands. We suggest you to proceed as follows:

- Read the chapter about the *Configuration File Format*.
- Check the *examples* folder and have a look at the sample configuration files. They showcase various of Padricks capabilites.
- Read the chapter *Generated Hardware Overview and Integration* to get a better understanding of the RTL that padrick generates and how to integrate it in your SoC project.
- Have a look at the RTL that padrick generates from the example YAML files to better understand the structure of the generated pad multiplexer
- Check the options available with the various CLI commands (either *online* or directly in your terminal with the *-h* option).
- Once you have your configuration ready, have a look at the generated source code.
- In case something is unclear, state your question on Github Discussions Forum
- If you find a bug or want to request file an issue or if you already have a solution, file a pull-request.

1.3 Padframe Configuration File

In the getting started guide you got a first grasp on how to use Padrick and already saw a brief example of a configuration file that tells padrick what to generate. In this chapter, we will deep dive into the configuration file syntax. After you familiarized yourself with the basic Padrick config structure, have a look at the **`exhaustive syntax reference <Syntax Reference_>`_**.

1.3.1 Concepts and Terminology

The basic idea behind padricks configuration file structure is to separate the specification into three different parts:

- A technology dependent section that defines the IO cells and its configuration signals
- An IO peripheral dependent section that defines the peripheral signals present in this SoC
- A chip specific mapping that glues the above two domains together

This separation of concerns allows to easily port an existing SoC to a new technology by only adapting the technology dependent configuration file section. Similarly, reusing same peripherals in a new SoC with different numbers of pads and multiplexing scheme but same target technology just requires a change of the mapping sections while the technology specific section and parts of the IO peripheral section can be reused.

To better understand the remaining parts of this chapter, lets first start with some terminology:

Pad:
RTL instance of an IO pad. Each Pad instance will typically have several pad signals to control the funcionality of the TX Buffer, driving strength and the actual pad -> SoC and SoC -> pad signals.

Port:
Throughout this document and in the scope of padrick, a Port denotes a set of signals from an SoC peripheral that can be routed to *one* of the available pads within the same Pad Domain. E.g. an I2C peripheral would expose the I2C_SCL and I2C_SDA port. The Port I2C_SDA for example could consist of the *Port signals* sda_tx, sda_rx and out_enable which all correspond (through some logic mapping) to the pad signals of a single IO pad.

1.3.2 Configuration File Syntax

The configuration file is written in YAML syntax. If you are unfamiliar with YAML or only sporadically used it so far, please take 2-3 minutes to read up on its most important featues since this will allow you to write cleaner configuration files. Especially the "anchor" and "reference" feature is quite usefull for this particular tool since it avoids copy-paste hell.

Note:

The YAML engine in padrick supports inclusions of external files to
modularize your config file. You can e.g. have a common config file to define the peripherals in your system that you combine with your technology specific YAML file. The examples folder contains an example config file that showcases the feature.

At the root, the configuration file contains three key-value pairs:

- name: The name of the pad_frame to generate (usefull if there is more than one and you want to avoid naming collision of the generated RTL)
- manifest_version: The current configuration file syntax version used for this particular file (at the moment, this is always the value 1)
- pad_domains: A yaml list of Pad Domains (see next subsection)

1.3.2.1 Pad Domains

A Pad Domain incorporates a collection of IO Pads, corresponding configuration registers and multiplexing logic. Pad Domains do not interact with each other and are generated in individual RTL modules for simplified Power Intent description (i.e. Power Gating of High-performance IO pads).

Pad Domains also define the scope of IO signal to IO pad routing. Every pad_domain contains its own crossbar that by **default** allows mapping of every *Port* to every *Pad* within the same pad domain. In combination with the possibility to define staticly controlled pads (pads that are controlled by one set of external signals only) this approach allows to map any existing PULP SoC padframe multiplexing scheme and much more advanced ones to the configuration file. While for many SoCs a single pad domain might be enough, multiple pad domains are most usefull for chips where IO pads are partially power gated. The clear separtion between pad domains makes it very easy to specify the power intent for such power gating schemes.

Each pad_domain in the configuration file contains 3 entries: - A list of pad_types (technology specific) - A list of pad_instances (technology agnostic but chip specific) - A list of Port Groups (technology and chip agnostic)

Declaration of IO Cells (Pad Types)

A Pad Type defines an available pad cell from your IO cell library that your are going to use in your design. One design might use several differ IO pad cells e.g. low-power ones and high-speed pads, pads with integrated pull-down or pads dedicated for differential signalling. Each `pad_type` is characterized by it's `name` and optional `description` key and most importantly the `template` key:

```
pad_types: # This section contains a list of pads
- name: pull_down_pad # user defined name of the pad. Used to reference it
                      # in the pad_list
  description: TSCM65 pad with controllable integrated 1kOhm pull down resistor
  # The template value is a Mako template (https://www.makotemplates.org/)
  # that specifies how to instantiate this partuclar pad. The '|' in the
```

(continues on next page)

(continued from previous page)

```
  # beginning is YAML syntax to allow multiline strings without the need
  # for manual character escaping
  template: |
    PDDW04808 ${instance_name} (
    .PAD(${conn["pad"]}),
    .IE(${conn["enable_rx"]}),
    .DS(${conn["driving_strength"]}),
    .I(${conn["chip2pad"]}),
    .O(${conn["pad2chip"]})
    );
  pad_signals:
    .... # See below for example
```

The Instantiation Template

The template is a multiline string that describes how this particular pad shall be instantiated within the autogenerated RTL in the form of a *Mako* template. While most of the string will probably be just a single SystemVerilog Module instantiation it also contains special markers that the *Mako* template library will replace with the appropriate content. The syntax of these template markers is quite simple if you are already familiar with Python. Check the quickstart guide on their webpage for more information https://www.makotemplates.org. For the sake of understanding the above examples it suffices to know that `${...}` is special Mako syntax to mark a python expression. When rendering a Mako template, the template render function is supplied with some user variables which are then available in the scope of such expression markers. The template render function will evaluate the python expression and replace the marker with the expressions value during template.

During instantiation of the pads, padrick renders each template by supplying it with two python variables that can be referenced within the Mako markers:

`instance_name`:
: A string containing the instance name that should be used for this particular instantiation of the Pad Type.

`conn`:
: A dictionary containing the wiring signals corresponding to the declared Pad Signals for this Pad Type that should be connected to this IO Pad during instantiation. E.g. when rendering the instantiation of IO pad "pad_gpio3", ${conn[enable_rx]} will be replaced with something like `s_pad_gpio3_rx_en` which is an autogenerated internal SystemVerilog wiring signal.

The `conn` variable is used to connect the wiring signals to your IO pad during instantiation. You can define arbitraring IO cell wiring signals in the `pad_signals` section of your `pad_type` configuration (see next section).

Pad Signals

Padrick must not only know how to instantiate your pads, it must also be aware of all pad config signals like tx buffer enable, driving strenths, i/o signal, landing pads etc. Padrick does not contain a list of hardcoded IO config signals but leaves full control to the user.

Each pad_type has a set of associated pad signals that are required to control the pad. For a typical IO pad, there are at least three signals:

- The signal connecting to the pads TX-buffer (SoC -> pad signal)
- The signal connecting to the pads RX-buffer (pad -> SoC signal)

- The pad signal itself which connectes to the toplevel of the RTL and is wired to the bonding pads/bumps of the chip. In addition to these signals there are most often numerous additional signals that control additional features of the pad like driving strength, optional schmidt-triggers etc.

Here is a (well documented) example of a `pad_signals` section for a very rudimentary IO pad:

```
pad_signals: &default_pad_signals #This is a YAML anchor to reuse a subblock somewhere␣
↪else. Use it to avoid copy paste hell!
# The pad signals section specifies a list of all pad signald used in
# this particular pad_domain. This include the rx signal, tx signal, the
# actual pad signal as well as all pad configuration signals. These are
# the signals that can be referenced by name in the pad instantiation
# templates within the pad_types sections, the connections of each pad
# within the pad_list as well as the connections section in the
# port_list.
  - name: output_en
    description: "Enables the output driver of the pad" #optional description
                                                        #of the signal
    size: 1 # The number of bits
    kind: input # The signal is an input signal to the pad i.e. a signal
                # driven by the chip that controls the pad.
    conn_type: dynamic # This means, the signal value is dynamic. It can
                       # either be controlled by an autogenerated
                       # configuration register or (at runtime
                       # configurable) an IO signal (if any IO signal
                       # within the pad_domain is referencing it).
    and_override_signal: s_enable_all_outputs # Optional override signal
                                              # that is and-gated with
                                              # the control signal
    default_reset_value: 0 # The default reset value of the pad signal
                           # if not overriden in the "connections"
                           # section of a particular pad instance
  - name: driving_strength
    description: "Driving strength of the output driver"
    size: 3
    kind: input
    conn_type: static # This means, the signal has a static value that
                      # is either driven by a single signal or a
                      # constant value. The difference between the
                      # dynamic type is, that this pad_signal is not
                      # arbitrary connectable with IO signals in a
                      # crossbar fashion but tied to one dedicated
                      # signal only. The actual signal or value assigned
                      # is defined individually for each pad in the
                      # padlist or described globally with the
                      # default_static_value.
    default_static_value: 0 # The default static value of the signal if
                            # not overriden in the "connections" section
                            # of a particular pad instance
  - name: enable_rx
    description: "Input buffer enable"
    size: 1
    kind: input
    conn_type: dynamic
```

(continues on next page)

(continued from previous page)

```
    default_reset_value: 1
  - name: pad2chip
    description: "The signal that connects to the pads RX buffer"
    size: 1
    kind: output
    conn_type: dynamic # In case of static output pad_signals, literal
                       # value assignments are illegal since the signal
                       # is not drivable from the outside. Only the name
                       # for a dedicated padframe output signal can be
                       # specified.
  - name: chip2pad
    description: "The signal that connects to the pads TX driver"
    size: 1
    kind: input
    conn_type: dynamic
    default_reset_value: 1
  - name: pad # The name of the signal can be chosen arbitrarily but for
              # the actual pad signal (the signal that connects to the
              # bonding pads) the name "pad" is a good choice. It is
              # legal to specify more than one signal of type pad (e.g.
              # if you want to instantiate a special differential
              # signaling pads ). However, at least one siganl of kind
              # pad is required
    size: 1
    kind: pad # Pad signals are handled specially. They are always exposed
              # directly to the toplevel of the generated padframe module and no
              # connection type, override signals or default values are allowed.
```

Pad Signal `kind`

Padrick does not make any assumption about the particular features controlled by a pad signal and does not do a distinction between the actual I and O signal or configuration signals. Padrick knows only three *kinds* of padsignals:

input:
: Signals that are inputs to the pad_instance cell e.g. chip->pad signal or driving_strength signal

output:
: Signals that are outputs to the pad_instance cell. E.g. pad->chip signal or power_up_ack signal.

pad:
: Signals that correspond to a bonding pad and should be routed to the toplevel of the RTL. While typically an IO pad contains only one Pad signal of this kind, padrick can perfectly handle pads with more than one landing pad signal (e.g. for differential signaling pads).

Connection Types

The connection type of a pad_signal determines, whether this particular signal is later-on to be controlled statically or dynamically.

Pad signals of type static do not have an input-multiplexer and thus cannot be controlled by the routable Port signals. Instead, they are either tied to a constant logic level (e.g. 1'b0 to tie it to zero) or a logic expression of external signals consisting of unary or binary operators and signal identifiers. This connection type is usefull for pad_signals you don't need to control at runtime but should be hardwired to instance specific values or connected to external signals. E.g. a static pad signal could be controlled by a single external signal e.g. "~``global_power_down``" connected to RX_en and TX_en.

For each **dynamic** pad signal, a configuration register is auto-generated for **every pad instance**. This provides the user with control over the signal in the default case, where no *Port* is routed to this particular pad instance's pad_signal. Thus, pad_signals of type dynamic can be controlled by all connectable (more on how to control connectivity in chapter *Port Multiplexing*) ports within the same pad_domain that referencec them. In other words; if you connect some port (e.g. `I2C_SDA`) to your pad instance, that port might take over control over the `output_en` and `enable_rx` pad signals. Other dynamic pad_signals like a `schmitt_trigger_en` are not controlled by the I2C peripheral. In such a case (and also if no port is connected to the pad instance at all) the pad signal is driven (for signals of `kind: input`) or accessible (for signals of `kind: output`) via the auto-generated config register file.

Dynamic pad_signals of kind `input` require you to specify a default_reset_value for the auto generated register. If not overriden during pad instantiation, the value you specify here will become the reset value of the corresponding configuration register. On the other hand, static pad_signals of kind `input` require you to specify a `default_static_value`; a static expression connected to the pad_signal if not overriden during pad instantiation.

Pad Instance List

The pad list contains a list of concrete pad instantiations. This is the place where you actually define, how many pads there are within your design. Each pad instance specifies a name for the pad, references the particular Pad Type to use (you might have multiple IO cell flavors to choose from) and a *static signal connection list*.

Here is an example:

```
pad_list:
  - name: pad_ref_clk # The instance name of the pad.
    description: "32kHz reference clock" #Optional description of the pads function
    pad_type: pull_down_pad
    is_static: true # Declaring a whole pad as static overrides every single
                    # pad signal's conn_type for that particular pad
                    # instance to "static".
    connections: # A list of static pad signal connections (for static
                 # signals) or default config register values (for dynamic
                 # pad signals)
      pad2chip: ref_clk
      chip2pad: ~ #Leave unconnected, only legal for pad signals of kind
                  #"input"
      enable_rx: 1 #pad signals of kind "input" any SystemVerilog literal is
                #valid.
      driving_strength: 0
  - name: pad_gpio
    description: "General Purpose Input and Outpud pads. These pads can be configured to␣
↪connect to any peripheral pad port."
    multiple: 32 #Generate 32 instances of this pad. Each instance will have
```

(continues on next page)

(continued from previous page)

```
                #its instance name postfixed with the index
                #number.
    pad_type: pull_down_pad
    is_static: false #False is the default value, thus explicitly specifying
                     #static as false is optional. With this option, each
                     #pad_signal assumes the declared conn_type.
  - name: pad_high_speed
    description: "High-speed IO pads for fast IO signals. "
    multiple: 10
    pad_type: high_speed_pad_gf22
    mux_group: hs_pads # An optional string that specifies a custom multiplexing
                       # group. All pads and ports within the same pad_domain
                       # and multiplexing group can be connected to each other.
                       # Default value: "all" By default all pads and ports
                       # end-up in the "all" multiplexing group and thus by
                       # default, every port can be connected to every pad
                       # within the same domain.
```

Hint: You will learn more about generating multiple pad/port instances in *Generating Multiple Ports/Pads with Regular Structure*.

Static Signal Connections and Config Register Reset Values

For each pad instance, the user can supply a `connections` list entry that overrides how static pad signals of this particular pad instance are to be connected or what the reset value of the corresponding configuration register shall be. The `connections` field contains a mapping of *Pad Signal names* to *expressions*. The pad signal name is just a reference to a **static** or **dynamic** pad_signal declared for the chosen pad type. The expression on the other hand must be a simplified subset of a SystemVerilog expression.

Expression may consist of simple SystemVerilog literals (e.g. `45`, `8'h0a`, `'0` etc.), unary and binary operators and signal identifiers without subscripting (e.g. `out_en_i` is legal, `out_en[45]` *is not legal*).

For pad_signals of kind ~output~ only single signal identifiers or the empty expression are allowed. After all, an output signal cannot be connected to an expression.

For dynamic pad_signals only constant expressions are allowed since this value is used as the reset value when asynchronously resetting the auto-generated register file.

The pad instance will be wired with the supplied expression and the generated pad_frame SystemVerilog module will expose each static signal used within any of the expressions within the pad_domain in its port_list for the user to connect these signal with the appropriate SoC logic. E.g.:

```
...
  connections:
    pad2chip: scan_en_i
    chip2pad: ~ # '~' is YAML syntax for 'None'. In this context it means leave
                # the signal unconnected, only legal for pad signals of kind
                # "input"
    enable_rx: 1 #pad signals of kind "input" any SystemVerilog literal is
                 #valid.
```

(continues on next page)

(continued from previous page)

```
    enable_tx: ~test_en_i & gpio1_en_rx
...
```

This will cause the pad_frame to expose the signals `scan_en_i`, `test_en_i` and `gpio1_en_rx`.

The direction and size of each of those static signals is inferred from the size and directionality defined for the particular pad_signal they are connecting to. If a static signal (signal identifiers on the right-hand-side of the connections list) is used in expressions for multiple pad_signals with different sizes an error is issued since size inference would be ambiguous.

Static signal identifiers with identical name within the same pad_domain denote the same signals. Thus if you have several pad instances whith connections entries like:

```
connections:
  enable_rx: input_buffers_en_i
```

They will all be connected to the same input signal `input_buffers_en_i`.

Ports and Port Groups

Port groups provide logical grouping of *ports* and *peripheral signals* which are muxed on you pad instances. Peripheral signals are the signals your IO facing peripheral exposes (e.g. `i2c_sda_tx_en` or `uart_rx`). *Ports* on the other hand are roles assigned to an IO pad when muxed to it. A *port* might make have to make use of multiple peripheral signals when it is connected to a pad. E.g. when connecting an I2C `sda` port to some particular pad, you need not only to connect the `i2c_sda` signal to the pad but also some `i2c_sda_tx_en` to control the pads directionality. The **ports** within a port group thus need to specify a logical mapping between peripheral signals and the **pad signals** defined in the `pad_types` section.

Hint: Static pads define their connected signals directly, see *static signal connections and config register reset values*

A concrete example should make things clearer. Here we define a port group for an I2C peripheral which consists of two ports (SDA and SCL):

```
port_groups:
  - name: i2c_0
    mux_groups: [all] # You will learn about mux_groups in the next section.
    output_defaults: 1'b0
    ports:
      - name: i2c_scl
        description: "Bidirectional I2C clock signal"
        connections:
          chip2pad: scl_out
          scl_in: pad2chip
          enable_tx: ~oen & i2c_en #You can use verilog expression combining multiple␣
↪peripheral signals in your connections
          enable_rx: oen & i2c_en

      - name: i2c_sda
        description: "Bidirection I2C data signal"
        connections:
          chip2pad: sda_out
```

(continues on next page)

(continued from previous page)

```
        sda_in: pad2chip
        pull_up_en: 1'b1 # You can also use literals if e.g. I2C pad requires pull-ups␣
↪to beautomatically enabled if I2C_SDA is connected to a pad.
        enable_tx: ~oen & i2c_en #You can use verilog expression combining multiple␣
↪peripheral signals in your connections
        enable_rx: oen & i2c_en
```

Each `port_group` must be defined with a `name`, some optional `description` and a list of ports (we will elaborate more on the `mux_groups` key in chapter *Port Multiplexing*). Each port again is defined with a name and optional description and a `connections` block. The connections block tells padrick how to connect the peripheral signals to the target pad when the user configures the port to be connected to a particular pad (muxing configuration registers). The individual connections can be read like assignments i.e. the signal on the left-hand-side is assigned the value of the expression on the right-hand-side. The identifiers used are either `pad_signal` names or *implicitly* defined peripheral signals.

Considering the example I2C port group above. Let's assume connected (by writing to the auto-generated config register) the `i2c_scl` port to some pad `mypad_08` whose pad instance uses the same `pad_signals` as defined in our earlier example. In that case the connection block instructs padrick to connect mypad_08's `chip2pad` signal to the `scl_out` peripheral signal. The IO pads `pad2chip` drives the peripheral signal `scl_in`. The `enable_tx` pad_signal is driven with a logic expression that consists of the two peripheral signals `oen` (an active low output-enable) and `i2c_en` (some global peripheral enable signal). The right hand side of a port connection can also be a literal e.g. if certain pad configuration signals should be tied to constant values when the peripheral is connected to the pad (e.g. `enable_tx: 1'b0` if a port is always an output as would be the case for `uart_rx`).

Important: Note that we didn't explicitly define our *peripheral signals* anywhere. Merely specifying a signal name in the connections block of a port implicitly defines the peripheral signal and causes padrick to generate the necessary module ports and muxing logic in the generated pad multiplexer. *The scope of the implicitly defined peripheral signals is the whole port group.* Thus in our above example, the `enable_rx` signals used in the ports `i2c_scl` and `i2c_sda` reference the exact same signal. Therefore, peripheral signals may be shared amongst ports within the same port group.

From the example before it should have become clear, that your peripheral can control any pad signal you defined for your `pad_type`. If you're peripheral needs to control driving strenghts, schmidt-triggers or whatever control signal your IO library exposes this is all possible. The more interesting question is however, what happens with the pad signals that you're *port* does **not** use? E.g. we didn't specify a connection for the `driving_strenght` signal. What driving strenght is used when our `mypad_08` is used as `i2c_scl` port? The answer is pretty simple:

Important: Every (dynamic) `pad_signal` that is not mentioned in your *port's* connection block will be controlled by an auto-generated pad configuration register whose reset value is specifed in the pad instance's `connection` block (see *static signal connections and config register reset values*). E.g. since we did not specify any connection for the `driving_strength` signal, the driving strenght of `mypad_08` will remain controlled by `mypad_08`'s pad configuration registers.

1.3.3 Generating Multiple Ports/Pads with Regular Structure

Generating pad instances or ports of a regular structure can become quite verbose if every instance is explicitly described in the YAML config file. Therefore, Padrick contains a feature for templated vectorization of **pad instances**, **port_groups** and **ports**. Each of these entities accepts the optional `multiple` key to instruct Padrick to generate multiple copies of the entity. During vector expansion, padrick looks for special text markers containing a mini expression language to generate the names, descriptions etc. of the vectorized entity. An example should make the explanation much easier:

```
pad_list:
  - name: gpio{i:2d}
    description: "GPIO No {i}"
    is_static: false
    pad_type: high_speed_pad_gf22
    multiple: 32
```

While parsing the config file, padrick will expand this vectorized pad_instance to 32 copies. Padrick will replace the name of each pad with `gpio00`, `gpio01` until `gpio31`. The description is handled similarly.

1.3.3.1 Mini Expression language

During expansion of the vectorized entity, padrick scans `name`, `description`, `mux_groups`, `connections` etc. for occurence of mini expressions (e.g. `{i:2d}`).

Each mini expression has the following format:

`{<expression>:<format>}` or `{<expression>}` (if you want to use the default format `d`)

`expression` can be any expression consisting of:

- the binary operators '+', '-', '*' (multiply), '/' (integer divide), '%' (modulo)
- the unary operators '+', '-'
- braces '()' to indicate associativity
- integer literals
- the loop variable `i` (a variable that starts counting from 0 during vector expansion and increments by one for every instance copy).

E.g.

```
name: gpio{i/2}_{i%2+1}
multiple: 4
```

Will be expaned to `gpio0_1`, `gpio0_2`, `gpio1_1` and `gpio1_2`.

The format specifier consists of `[<lenght>]<format_class>`.

Format Class `d`:
Format result of the expression in decimal representation. The optional `length` specifies the amount of **zero padding**.

Format Class `o`:
Same as `d` but format expression in octal representation.

Format Class `b`:
Same as `d` but format expression in binary representation.

Format Class x:
Same as d but format expression in hexa decimal representation.

Format Class c:
Format expression in Base26 and map the individual 'digits' to the lowercase letters of the latin alphabet. Supplying the optional `length` forces padding with the letter *a*. E.g. `pad_{i:c}` will be mapped to `pad_a`, `pad_b`, `pad_c`, ..., `pad_aa`, `pad_ab`, `pad_ac` and so forth. E.g. `pad_{i:2c}` will be mapped to `pad_aa`, `pad_ab`, `pad_ac` etc.

Format Class C:
Same as `c` but use upper-case letters.

Here is another example:

```
name: pad_{i/4:C}{i%4:2d}
```

Expands to `pad_a00`, `pad_a01`, `pad_a02`, `pad_a03`, `pad_b00`, `pad_b01` etc.

Hint: You can use the padrick command `padrick config <your_padrame.yml>` to parse the config file and print it in expanded form. This will resolve all cross links in your config file (e.g. references to `pad_types`) and will expand all vectorized `port`, `pad_instance` etc. This is quite helpfull to debug how padrick is treating your vectorized config files.

1.3.4 Port Multiplexing

By default, Padrick allows routing any *Port* to any (non-static) *Pad Instance*. However, the degree of routability can be adjusted very finely. Padrick uses so called *mux groups* to configure the connectivity between ports and pad instances. Every pad instance and every port is a member of *one or several* mux groups. Ports can be dynamically connected to all pad instances which are contained in any of the port's mux groups. I.e. `port_xy` can be connected to `pad_123` if `pad_123` is part of one (or multiple) of `port_xy`'s mux groups. In more mathematical terms; Each pad_instance and each port specify a set of labels (mux_groups), whenever there is set-intersection between a pad_instance and a port, they can be connected with each other.

A mux group is denoted by a simple string identifier and declared with the *mux_groups* key in the config file. E.g. the following config snippet declares a pad called `my_pad` that is member of the mux_groups `mux1`, `my_pads`, and `all`:

```
pad_list:
  - name: my_pad
    mux_groups: # some examples use the more compact notation [mux1 my_pads self]. Both␣
↪styles are valid YAML lists.
      - mux1
      - my_pads
      - all
    connections:
      ...
```

Similar to *peripheral signals* or *static connection signals* you don't have to explicitly declare *mux groups*. The first usage of an identifier creates the new mux group. You can use any C-identifier-like string as the mux group name.

You probably noticed, that our previous config example snippets most of the time did not specify the `mux_groups` key. The key is optional and has the default value `[all]`. I.e. by default, all ports and all pads are member of a mux group called `all`. If you followed our explanation so far you should realize now, why by default, all ports can be connected to all pads with this default value.

Apart from ports and pad instances, mux_groups can also be applied to a complete port group. In that case the declared mux_group acts as a default for any port within the port group that doesn't explicitly specify its own port group.

Lets have a look at small example with a couple of pads and a couple of ports:

```
pad_list:
  - name: pad1
    mux_groups: [mx1]
    ...
  - name: pad2
    mux_groups: [mx1, mx2]
    ...
  - name: pad3
    mux_groups: [mx2]

port_groups:
  - name: spi
    mux_groups: [mx1]
    ports:
      - name: sck
        mux_groups: [mx2]
        ...
      - name: mosi
        mux_groups: [mx1, mx2]
        ...
      - name: miso
        ... # No mux_groups specified for mosi thus the port_group's default (mx1) applies
```

In this small example, we used 2 different mux groups called `mx1` and `mx2`. We have the following connectivity for the 3 ports:

- Port `sck` can be connected to `pad2` and `pad3` since both are member of the `mx2` group.
- Port `mosi` can be connected to all three pads since all pads are member of either `mx1` or `mx2`.
- Port `miso` does not specify a mux group, thus the default value of the *mux group* applies (if the mux group doesn't specify one, `[all]` is used). Therefore, `miso` can be routed to either `pad1` or `pad2`.

1.3.4.1 Mux Group Templating

Combining this chapter with the knowledge from mini expression language we now have all the ingredients to define more complex IO multiplexing schemes. The key realization is, that mux_groups can be templated using the mini expression language like we templated the port/pad instance names and descriptions in the examples on *generating multiple ports/pads with regular structure*.

Lets consider the following example:

```
...
pad_list:
  - name: hs_pad{i}
    multiple: 4
    pad_type: highspeed_pad
    mux_groups: [hs_pads, hs_pad{i}]
    ...
  - name: ls_pad{i}
```

(continues on next page)

(continued from previous page)

```
    multiple: 4
    pad_type: lowspeed_pad
    mux_groups: [ls_pads, ls_pad{i}]
    ...

port_groups:
  - name: hs_gpio
    ports:
      - name: gpio{i}
        multiple: 4
        mux_groups: [hs_pad{i}]
  - name: ls_gpio
    ports:
      - name: gpio{i}
        multiple: 4
        mux_groups: [ls_pad{i}]
  - name: i2c
    mux_groups: [ls_pads]
    ports:
      ...
  - name: hyerflash
    mux_groups: [hs_pads]
    ports:
      ...
```

In this example, we have instantiate 4 highspeed (hs) and 4 low speed (ls) pads. After vector expansion the pad `hs_pad0`, will be member of the mux_group `hs_pads` and `hs_pad0`, the pad `hs_pad1` will be member of mux groups `hs_pads` and `hs_pad1` and so forth.

On the port side, we declare a low-speed gpio port group, a high speed gpio port group, an i2c port group and a hyperflash port group.

Since the individual ports of a GPIO peripheral are usually all identical, it doesn't make much sense to waste the routing resources to allow routing e.g. `GPIO0` to `pad4`, you just use `GPIO4` instead. To allow for such a routing scenario, each port in the `hs_gpios` port group is member of the corresponding pad's mux group. E.g. port `gpio0` of the `hs_gpio` port group is member of the `hs_pad0` mux group, port `gpio1` is member of `hs_pad1` and so forth. This results in the intended scenario. Since the i2c port group specifies the default mux group `ls_pads`, every port within `i2c` can be routed to any of the 4 low-speed pads, while any port of the hyperflash peripheral can be routed to any of the high_speed pads.

1.3.4.2 Default Pad Roles

By default, after reset each `pad_instance`'s set of dynamic pad signals is fully controlled by the auto-generated configuration register file. I.e. in order to have your dynamic pads configured as inputs after power-on reset, you have to choose the right reset values in your pad_instance's `connections` block. However, sometimes you want a pad multiplexing scheme with a default `port` to `pad_instance` assignment right after reset. For these cases, you can use the optional *default_port* key when declaring a pad_instance to assign it a default port. The port name is specified using dot-notation, i.e. `<expanded_port_group_name>.<expanded_port_name>`.

Here is an example:

```
- name: pad_05
  pad_type: pull_down_pad
```

(continues on next page)

(continued from previous page)

```
  default_port: hs_gpio.gpio05
```

For multi pads, i.e. pads with a `multiple` field larger than 1, the situation is a bit more complex. Since Padrick `v0.3.4`, you can also supply a dictionary of evaluated mappings to specifiy individual default_ports for multi-pads. An example makes this much easier to understand:

```
- name: pad_io{i}
  pad_type: pull_down_pad
  multiple: 32
  default_port:
    '*': gpio.gpio{i}
    pad_io5: uart.tx
    pad_io6: uart.rx
    pad_io16: spi.sck
    pad_io17: spi.mosi
    pad_io18: spi.miso
```

This example snippet in the `pad_list` section defines 32 io pads with the expanded names `pad_io0`, `pad_pio1`, ..., `pad_io31`. The second entry in the `default_port` matches with instance `pad_io5` and assigns it the default port `uart.tx`. Like for most fields in padrick, the port specifier string can contain miniexpressions and will be expanded accordingly. The entries are applied in the order they are listed and can override each other. This behavior is leveraged by the first entry; it uses the wildcard padname `'*'` which matches with every expanded pad. Thus every `pad_io<xy>` instance will be assigned the default role `gpio.gpio<xy>`. However, since the other entries are listed afterwards, they override this default assignment.

Important: The order of the `default_port` mapping matters since the mappings can override each other. If you use the wildcard match entry `'*'`, make sure it is the first entry in the list.

1.3.5 Config File Schema

The following table contains an auto-generated schema reference of the configuration file format.

1.3.5.1 Padframe Config

Padframe class that represents the padframe configuration parsed from the configuration file. **Attributes:** manifest_version (int): The manifest version used by the parsed configuration file. name (str): Name of the pad_frame module. description (str): An optional short description of the padframes. pad_domains (List[PadDomain): A list of PadDomains within this padframe.		
type	*object*	
properties		
• **manifest_version**	*Manifest Version*	
	type	*integer*
• **name**	*Name*	
	type	*string*
	pattern	^[_a-zA-Z](?:[_a-zA-Z0-9])*
• description	*Description*	

continues on next page

Table 1 – continued from previous page

	type	*string*			
• **pad_domains**	*Pad Domains*				
	type	*array*			
	items				
	•	#/definitions/PadDomain			
	minItems	1			
• user_attr	#/definitions/UserAttrs				
definitions					
• PadSignalKind	*PadSignalKind*				
	An enumeration.				
	type	*string*			
	enum	input, output, pad			
• ConnectionType	*ConnectionType*				
	An enumeration.				
	type	*string*			
	enum	static, dynamic			
• UserAttrs	*UserAttrs*				
	type	*object*			
	additionalProperties	anyOf	•	#/definitions/UserAttrs	
			•	type	*integer*
			•	type	*boolean*
			•	type	*string*
• PadSignal	*PadSignal*				
	type	*object*			
	properties				
	• **name**	*Name*			
		type	*string*		
	• size	*Size*			
		type	*integer*		
		maximum	32		
		minimum	1		
		default	1		
	• description	*Description*			
		type	*string*		
	• **kind**	#/definitions/PadSignalKind			

continues on next page

Table 1 – continued from previous page

	• conn_type	#/definitions/ConnectionType
	• and_override_signal	*And Override Signal*
		type: *string*
		default:
	• or_override_signal	*Or Override Signal*
		type: *string*
		default:
	• default_reset_value	*Default Reset Value*
		type: *integer*
	• default_static_value	*Default Static Value*
		type: *string*
	• user_attr	#/definitions/UserAttrs
	additionalProperties	False
• PadType	*PadType*	
	type	*object*
	properties	
	• **name**	*Name*
		type: *string*
		pattern: ^[_a-zA-Z](?:[_a-zA-Z0-9])*
	• description	*Description*
		type: *string*
	• **template**	*Template*
		type: *string*
	• pad_signals	*Pad Signals*
		type: *array*
		default: []
		items
		• #/definitions/PadSignal
	• user_attr	#/definitions/UserAttrs
	additionalProperties	False
• Port	*Port*	
	type	*object*
	properties	
	• **name**	*Name*
		type: *string*
	• description	*Description*

continues on next page

Table 1 – continued from previous page

		type	*string*		
	• connections	*Connections*			
		type	*object*		
		additionalProperties	type	*string*	
	• mux_groups	*Mux Groups*			
		type	*array*		
		default	['all', 'self']		
		items			
		•	type	*string*	
		minItems	1		
		uniqueItems	True		
	• multiple	*Multiple*			
		type	*integer*		
		minimum	1		
		default	1		
	• user_attr	#/definitions/UserAttrs			
	additionalProperties	False			
• PortGroup	*PortGroup*				
	type	*object*			
	properties				
	• **name**	*Name*			
		type	*string*		
	• description	*Description*			
		type	*string*		
	• mux_groups	*Mux Groups*			
		type	*array*		
		items			
		•	type	*string*	
		minItems	1		
		uniqueItems	True		
	• **ports**	*Ports*			
		type	*array*		
		items			
		•	#/definitions/Port		
	• output_defaults	*Output Defaults*			
		default	OrderedDict()		
		anyOf	•	type	*string*

continues on next page

Table 1 – continued from previous page

			•	type	*object*	
				additional-Properties	type	*string*
	• multiple	*Multiple*				
		type	*integer*			
		minimum	1			
		default	1			
	• user_attr	#/definitions/UserAttrs				
	additional-Properties	False				
• PadInstance	*PadInstance*					
	type	*object*				
	properties					
	• **name**	*Name*				
		type	*string*			
	• description	*Description*				
		type	*string*			
	• multiple	*Multiple*				
		type	*integer*			
		minimum	1			
		default	1			
	• **pad_type**	*Pad Type*				
		anyOf	•	type	*string*	
				pattern	^[_a-zA-Z](?:[_a-zA-Z0-9])*	
			•	#/definitions/PadType		
	• is_static	*Is Static*				
		type	*boolean*			
		default	False			
	• mux_groups	*Mux Groups*				
		type	*array*			
		default	[‘all’, ‘self’]			
		items				
		•	type	*string*		
		minItems	1			
		uniqueItems	True			
	• connections	*Connections*				
		type	*object*			
		additional-Properties	type	*string*		
	• default_port	*Default Port*				
		anyOf	•	type	*object*	
				additional-Properties	type	*string*

continues on next page

Table 1 – continued from previous page

			•	type	*string*
			•	type	*array*
				items	
				•	#/definitions/PortGroup
				•	#/definitions/Port
				maxItems	2
				minItems	2
	• user_attr	#/definitions/UserAttrs			
	additional-Properties	False			
• Pad-Do-main	*PadDomain*				
	A pad_domain contains the configuration about one collection of pads and ports that can connected with each other.				
	type	*object*			
	properties				
	• **name**	*Name*			
		type	*string*		
		pattern	^[_a-zA-Z](?:[_a-zA-Z0-9])*		
	• de-scrip-tion	*Description*			
		type	*string*		
	• **pad_types**	*Pad Types*			
		type	*array*		
		items			
		•	#/definitions/PadType		
		minItems	1		
	• **pad_list**	*Pad List*			
		type	*array*		
		items			
		•	#/definitions/PadInstance		
		minItems	1		
	• port_groups	*Port Groups*			
		type	*array*		
		default	[]		
		items			
		•	#/definitions/PortGroup		

continues on next page

Table 1 – continued from previous page

	• user_attr	*User Attr*				
		type	*object*			
		additional-Properties	anyOf	•	type	*string*
				•	type	*integer*
				•	type	*boolean*

1.4 Generated Hardware Overview and Integration

1.4.1 Architectural Overview

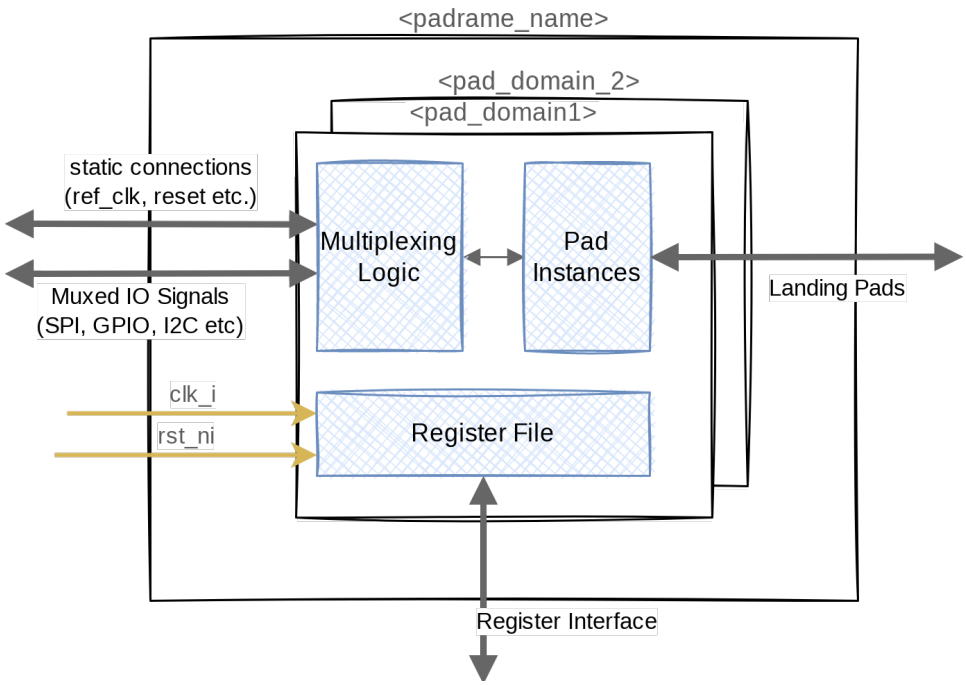

Fig. 1: RTL Architectural Overview of Padrick generated pad multiplexing IPs.

The padrick *generate rtl* command generates a complete RTL subproject including `Bender.yml` files for the Bender dependency management tool. The topleevel module `<padframe.name>.sv` wraps pad domain module instances, one for each pad domain defined in your config file. This structure simplifies power-intent specification in case you have multiple IO voltage/power domains. Each power domain contains a configuration register file which controls the IO Multiplexing logic and the default pad control signals. The multiplexing logic is wrapped in yet another, purely combinational submodule which contains multiplexers for the various peripheral port->pad signals and priority decoders + multiplexers for the pad->peripheral port direction. That is, writing to a pads multiplexer select signal config register will configure both directions of the pad <-> peripheral port connectivity.

Hint: In case multiple pads are configured to connect to the same peripheral port, e.g. `pad_01` and `pad_02` both

connect to port `i2c.sda`, the tx driver of both pads will be connected to the I2C peripheral's SDA_out signal but only `pad_01`'s RX buffer (GPIO with smaller index after alphabetic ordering) will connect to the I2C periphral's SDA_in signal. After all, we cannot connect two IO pads to the same input signal. This would cause a drive conflict.

At the same hierarchical level as the multiplexing logic module, there is a pad instantiation module that uses the pad template defined in the configuration file to instantiate the desired IO pads from the your IO library. It connects to the multiplexing logic module and is controlled by the configuration register file and the static connection signals exposed at the toplevel.

The toplevel module exposes the various signals of your padframe in the form of SystemVerilog structs. These are defined in the `pkg_<padframe.name>.sv`. In general, the generated toplevel padframe IP will expose the following signals:

`clk_i`
: The clock used for the configuration interface and the configuration registers (the multiplexing logic is purely combinational and thus unclocked).

`rst_ni`
: An active-low asynchronous reset signal used for resetting the generated internal configuration register.

`port_signals_soc2pad`
: A hierarchical struct of all `peripheral signals` with direction peripheral to IO pad organized by `port_group`.

`port_signals_pad2soc`
: A hierarchical struct of all `peripheral signals` with direction IO pad to peripheral organized by `port_group`.

Landing Pads
: A expanded list of bidirectional landing pad signals (i.e. the signals connecting to your IO landing pads or IO Bumps). Every pad signal of kind `pad` for every `pad_instance` is exposed in this list.

`config_req_i`
: The request side of a register_interface bus used to communicate with the internal configuration register files.

`config_req_i`
: The response side of a register_interface bus used to communicate with the internal configuration register files.

1.4.2 Generated Configuration Register File

As part of the RTL generator, padrick will auto-generate a configuration register file that is used to control the pad multiplexing the various IO pad control signals. Each register is 32-bit wide and exposed through a register_interface bus.

The tool internally leverages Opentitan's register tool (https://docs.opentitan.org/doc/rm/register_tool/) to generate those register file using an hjson configuration file. This intermediate hjson file generated by padrick is also available as part of the generated files of the RTL generator and provides the definitive and compact reference of all generated registers.

In general, two registers (or more if 32-bits are not enough) are generated for each *pad_instance*:

`<pad_instance_name>_CFG(0,1,2...)`
: This register controls the default values for each dynamic pad configuration signal that is not currently under the control of a connected peripheral. Each dynamic pad signal is assigned a dedicated field within this register with reset value controlled by the pad_instance's connection block. Here is an excerpt of a padrick generated register hsjon file:

```
{
        name: IOPAD_1_CFG
```

(continues on next page)

(continued from previous page)

```
        desc: '''
          Pad signal configuration.
        '''
        swaccess: "rw"
        fields: [
          {
              bits: "0"
              name: chip2pad
              desc: '''

              '''
              swaccess: "rw"
              hwaccess: "hro"
              resval: "0"
          },
          {
              bits: "1"
              name: tx_en
              desc: '''
                   Active high TX driver enable
              '''
              swaccess: "rw"
              hwaccess: "hro"
              resval: "0"
          },
          ...
        ]
    }
```

<pad_instance_name>_MUX_SEL

This register controls the multiplexing in front of the pad instance. Each *connectable* peripheral port is assigned an *enum value*. Here is an excerpt of a padrick generated register hjson file:

```
{
        name: IOPAD_1_MUX_SEL
        desc: '''
            Pad signal port multiplex selection for pad iopad_1. The programmed value␣
↪defines which port
            is connected to the pad.
        '''
        swaccess: "rw"
        hwaccess: "hro"
        resval: 0
        fields: [
            {
                bits: "2:0"
                enum: [
                    { value: "0", name: "register", desc: "Connects the Pad to the␣
↪internal configuration register. This is the default value."}
                    { value: "1", name: "port_SPIM_miso", desc: "Connect port miso from␣
↪port group SPIM to this pad." }
                    { value: "2", name: "port_SPIM_mosi", desc: "Connect port mosi from␣
```

(continues on next page)

(continued from previous page)

```
↪port group SPIM to this pad." }
                    { value: "3", name: "port_SPIM_sck", desc: "Connect port sck from␣
↪port group SPIM to this pad." }
                    { value: "4", name: "port_SPIM_cs", desc: "Connect port cs from port␣
↪group SPIM to this pad." }
                    { value: "5", name: "port_UART_rx", desc: "Connect port rx from port␣
↪group UART to this pad." }
                    { value: "6", name: "port_UART_tx", desc: "Connect port tx from port␣
↪group UART to this pad." }
                ]
            }
        ]
    }
```

Hint: Padrick calls OpenTitan's register tool internally. So although you have access to the padrick-generated hjson file you don't have to call reggen manually. Padrick takes care of generating the System Verilog RTL for the register file from the hjson internally.

1.4.3 Customization of Generated RTL/ Generating Custom Output Files

Padrick internally uses so called Mako templates (https://docs.makotemplates.org/en/latest/syntax.html) for RTL, driver etc. generation. The default template files embedded in Padrick's sourcecode should be fitting most needs. However, sometimes particular tape-out requirements require customization of the auto-generated RTL. A naive approach would be to just plainly modify the generated files. This approach is neither technology portable nor efficient since it requires the user to reapply the same modifications whenever the padframe structure (configuration YAML) changes and thus the RTL files need to be re-generated. Instead, Padrick provides you with the possibility to customize the Mako templates themselves to directly generate customized RTL.

1.4.3.1 Customizing Padrick Output with a Generator Settings File

Padrick's *generate* subcommands accept an optional flag (*-s <filename>*) to specify a `generator_settings.yml` file. This is a YAML file which allows you to control padrick's template rendering behavior. Here is an example customization file:

```yaml
manifest_version: 2
rtl_templates:
  toplevel_sv_package:
    name: SV package
    target_file_name: pkg_{padframe.name}.sv
    template: rtl_templates/pkg_padframe.sv.mako
    skip_generation: false
  pad_domain_top:
    name: Paddomain module {pad_domain.name}
    target_file_name: '{padframe.name}_{pad_domain.name}.sv'
    template: rtl_templates/pad_domain.sv.mako
    skip_generation: false
  pad_inst_module:
    name: Pad instantiation module {pad_domain.name}
    target_file_name: '{padframe.name}_{pad_domain.name}_pads.sv'
```

(continues on next page)

(continued from previous page)

```
    template: rtl_templates/pads.sv.mako
    skip_generation: false
driver_templates:
  ...
```

For each generator, there are various template customization entries. The key of the entry defines which output file is supposed to be customized. There are 4 different key value pairs you can specify for each entry:

name
: The `name` field is used for documentation and logging purposes only. It has no effect on the actual template rendering.

target_file_name
: This is a Mako template that renders to the filename of the generated file. E.g. in the example above, the RTL generator will render the `toplevel_sv_package` template using the filename: `pkg_<the name of your padrame>.sv`.

template
: The path to a mako template file used for rendering the output file. Here you can specify the path to your customized Mako template.

skip_generation
: If set to `true`, the output file for this template is not generated. Usefull, if you want to e.g. avoid generating the legacy IPApprox `src_files.yml` file.

Important: You *don't* have to write your modified template and the *generator_settings.yml`file from scratch. Padrick can generate a folder structure for you that already contains a `padrick_generator_settings.yml* file and a copy of each of the built-in templates. This makes customization much easier. Use the *padrick generate template-customization* command to create it.

1.4.3.2 Generating Custom Output Files

The generator settings file allows you to customize the output of existing padrick generators. However, you cannot add entirely new output formats. If you need to generate an additional file which padrick does not already have a dedicated generator for, you can use the generic template render command *padrick generate custom*. This command, in addition to you padframe_config.yml file accepts an additional mako template file argument.

Hint: With this command you can render your own custom templates and thus generate new output file formats without modifying padrick's source code. Still, if you wrote a template that might be of general interest (not a tape-out specifig output format) consider contributing it through a PR.

1.4.3.3 Writing Custom templates

As mentioned before, padrick uses the template rendering engine *mako* to create its output files. The advantage of mako over similar template rendering engines is, that it directly evaluates inline python expressions and thus allows very natural interaction between the template and a python data model.

Providing a tutorial on mako is outside the scope of this documentation. Please refer to https://docs.makotemplates.org/en/latest/syntax.html for more information. However, an important aspect of every template rendering flow is the variables available in the template rendering context, i.e. how do you access the padframe config data when generating the template. Padrick uses an advanced data modeling library called pydantic to validate your padframe configuration file and map it to a python class hierarchy. The mapped padframe configuration object (and instance of *padrick.Model.Padframe*) is directly exposed to your template's rendering context under the variable name `padframe`. Have a look at the built-in templates (use `padrick generate template-customization` to create a modifiable copy of them) on how to use this data model or inspect the python class documentation directly.

Hint: If you customize templates which are generated at the `pad_domain` level (i.e. one file is generated per pad_domain) the template, in addition to the `padframe` variable is handed a *PadDomain* instance under the variable name `pad_domain`.

1.4.4 HW Integration

Integration of the generated Padframe RTL is straigth forward:

1. In your toplevel module (or wherever you plan to instantiate the padframe), declare helper connection signals of struct type:
   - `pkg_<padframe.name>::port_signals_pad2soc_t`,
   - `pkg_<padframe.name>::port_signals_soc2pad_t`,
   - `pkg_<padframe.name>::static_connection_signals_pad2soc_t` and
   - `pkg_<padframe.name>::static_connection_signals_soc2pad_t`,
2. Connect all your peripheral signals and static conneciton signals to the helper struct signals.
3. Instantiate the `<padframe.name>.sv` module and connect it's port to your helper signals.
4. Connect your configuration bus to the the padframes configuration port. In case you are using a different protocol than `register_interface`, use one of the available protocol converters in https://github.com/pulp-platform/register_interface.

1.5 CLI Reference

This chapter contains the auto-generated documentation of the command line interface for padrick. The information you find here is identical to the info found in the CLI command help pages.

1.5.1 padrick

Generate padframes for SoC

```
padrick [OPTIONS] COMMAND [ARGS]...
```

Options

--version

Show the version and exit.

1.5.1.1 config

Print the parsed padframe configuration file

```
padrick config [OPTIONS] FILE
```

Options

-v, --verbosity <LVL>

Either CRITICAL, ERROR, WARNING, INFO or DEBUG

Arguments

FILE

Required argument

1.5.1.2 fusesoc-gen

Generator invocation for FuseSoC.

Parses the supplied config_file command and generates RTL + Core files in the current direcotry. Check the documentation for more information about available FuseSoC Generator parameters.

```
padrick fusesoc-gen [OPTIONS] CONFIG_FILE
```

Arguments

CONFIG_FILE

Required argument

1.5.1.3 generate

Generate various output files for the provided pad_frame configuration

```
padrick generate [OPTIONS] COMMAND [ARGS]...
```

Options

-s, --generator_settings_file <generator_settings_file>

A yaml file containing custom settings for the generate command.

constraints

Generate an SDC constraints file with set_case_analysis on all configuration registers of the padmultiplexer.

The generated SDC file is usefull in constraining the padmultiplexer to only consider a fixed static configuration to prevent STA from considering all possible multiplex configurations.

On top of the usual pad configuration file (CONFIG_FILE), this command accepts a case specification file as the second argument that specifies to which values the different multiplex registers shall be constrained to.

```
padrick generate constraints [OPTIONS] CONFIG_FILE CONSTRAINTS_SPEC_FILE
```

Options

-o, --output <output>

Directory where to save the SDC files

--header <header>

A text file who's content (extended with appropriate comment characters) is inserted as the header in each auto-generated file. Useful for copyright and author information.

--version-string, --no-version-string

Append current version of padrick to the header of each generated file.

Default
True

-v, --verbosity <LVL>

Either CRITICAL, ERROR, WARNING, INFO or DEBUG

Arguments

CONFIG_FILE

Required argument

CONSTRAINTS_SPEC_FILE

Required argument

custom

Render a user-specified custom Mako Template TEMPLATE file using the parsed CONFIG_FILE pad configuration data.

This command is usefull for any kind of desired output format for which Padrick doesn't already ship with the right template. The rendered template will be printed to OUTPUT. Both TEMPLATE and OUTPUT accept either a path to a file or the special argument '-' to read from/write to stdin/stdout.

```
padrick generate custom [OPTIONS] CONFIG_FILE TEMPLATE OUTPUT
```

Options

-v, --verbosity <LVL>

Either CRITICAL, ERROR, WARNING, INFO or DEBUG

Arguments

CONFIG_FILE

Required argument

TEMPLATE

Required argument

OUTPUT

Required argument

driver

Generate C driver to interact with the padframe.

```
padrick generate driver [OPTIONS] CONFIG_FILE
```

Options

-o, --output <output>

Location where to save the driver

--header <header>

A text file who's content (extended with appropriate comment characters) is inserted as the header in each auto-generated file. Useful for copyright and author information.

--version-string, --no-version-string

Append current version of padrick to the header of each generated file.

Default

`True`

-v, --verbosity <LVL>

Either CRITICAL, ERROR, WARNING, INFO or DEBUG

Arguments

CONFIG_FILE
Required argument

padlist

Generate a CSV file that lists all pads in your configuration.

```
padrick generate padlist [OPTIONS] CONFIG_FILE
```

Options

-o, --output <output>
Directory where to save the padlist CSV

-v, --verbosity <LVL>
Either CRITICAL, ERROR, WARNING, INFO or DEBUG

Arguments

CONFIG_FILE
Required argument

rtl

Generate SystemVerilog implementation from the padframe configuration.

```
padrick generate rtl [OPTIONS] CONFIG_FILE
```

Options

-o, --output <output>
Location where to save the RTL

--header <header>
A text file who's content (extended with appropriate comment characters) is inserted as the header in each auto-generated file. Useful for copyright and author information.

--version-string, --no-version-string
Append current version of padrick to the header of each generated file.

Default
`True`

-v, --verbosity <LVL>
Either CRITICAL, ERROR, WARNING, INFO or DEBUG

Arguments

CONFIG_FILE

Required argument

template-customization

Generate a padrick_gen_settings.yml file and folder structure containing copies of all internal Mako templates for customization of the generated file formats.

This is an advanced feature and allows the user to customize the internal Mako templates used to generate the various export files. In order for the customize option to have any effect, you need to invoke the generate commands with the additional -s flag: e.g.: padrick generate -s padrick_gen_settings.yml rtl my_padframe.yml

The -s option needs to come before the subcommand (in this case 'rtl').

```
padrick generate template-customization [OPTIONS]
```

Options

-o, --output <output>

Location where to save the RTL

1.5.1.4 install-completions

Install the command line tool's bash completion for your shell

If you don't provide any additional arguments this command tries to detect your current shell in use and appends the relevant settings to your .bashrc, .zshrc etc.

```
padrick install-completions [OPTIONS] [[bash|fish|zsh|powershell]] [PATH]
```

Options

--append, --overwrite

Append the completion code to the file

-i, --case-insensitive, --no-case-insensitive

Case insensitive completion

Arguments

SHELL

Optional argument

PATH

Optional argument

1.5.1.5 validate

Parse and validate the given config file

```
padrick validate [OPTIONS] FILE
```

Options

-v, --verbosity <LVL>
Either CRITICAL, ERROR, WARNING, INFO or DEBUG

Arguments

FILE
Required argument

1.6 License

modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or

for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

> To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "{}" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright {yyyy} {name of copyright owner}

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

> http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

1.7 Contributors

- Manuel Eggimann <manuel.eggimann@gmail.com>

1.8 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

1.8.1 Unreleased

1.8.1.1 Added

1.8.1.2 Changed

1.8.1.3 Fixed

1.8.2 v0.3.6 - 2022-12-14

1.8.2.1 Changed

- Prepare Metadata for Pypi release

1.8.3 v0.3.5 - 2022-12-14

1.8.3.1 Added

- Support for FuseSoC. Padrick now generated core files and can be invoked natively as a FuseSoC Generator.

1.8.4 v0.3.4 - 2022-12-06

1.8.4.1 Added

- User attributes now support nested dictionaries i.e. besides a string, boolean or an integer the value of an attribute can be another dictionary.
- The `default_port` field of pad_instances now support dictionaries of pad_name to port mappings. See the correspondingly updated `default_port` section in the docs.

1.8.4.2 Changed

- User Attributes of ports, port_groups and pad_instances are now expanded as well when instantiated multiple times. The same mini expression languages applies.
- Improved manifest version warning messages
- Bumped manifest version to 3.

1.8.4.3 Fixed

1.8.5 v0.3.3 - 2022-12-05

1.8.5.1 Added

- Added support for modular config file using pyyaml !include directives.

1.8.5.2 Changed

1.8.5.3 Fixed

1.8.6 v0.3.2 - 2022-10-21

1.8.6.1 Added

Added support for templated `default_port` in vectorized pad instances (pads with `multiple` > 1).

1.8.6.2 Changed

Removed now obsolete hint about `default_port` with vectorized pad instances compatibility.

1.8.7 v0.3.1 - 2022-10-19

1.8.7.1 Added

1.8.7.2 Changed

1.8.7.3 Fixed

- Regression in driver template rendering that caused an error during the register file generation.

1.8.8 v0.3.0 - 2022-10-18

1.8.8.1 Added

- Added read-only IP info register with version id and pad count values to RTL template

1.8.8.2 Changed

- Allow padframe generation without any muxed pads

1.8.8.3 Fixed

- Fix address width bug in address demux rules when generating padframes with power's of two number of registers

1.8.9 v0.2.0 - 2022-25-04

1.8.9.1 Added

- Added support for multiple multiplex groups per pad/port. Each pad/port/port_group can now be member of severa mux_groups. This causes the union of all specified groups to be muxable to the pad,port or port_group in question.
- Add support for multi-ports. Ports now accept the new optional *multiple* key that allows to specify several port with similar structure without copy-paste-hell
- Add support for templated names, description and mux_groups to multi-pads/multi-ports using the '{i}' token. This feature simplifies the definition of GPIO ports. Check the example config file for an example on how to use the feature.
- Add version flag to CLI to print version information
- Add validation of manifest version. The CLI now prints out an error message if the manifest version is not supported anymore with a help message which version of Padrick supports the out-dated manifest version.
- Add mini expression language for template tokens.
- Render assignment macros in systemverilog package to simplify hierarchical assignements of port groups.
- Add optional format code feature to index templates.
- RTL generate command now supports supplying file headers to insert into the auto-generated files
- Auto-generate SystemVerilog header files with struct assignment macros
- Added optional key *default_port* to pad_instances of the form "<port_group_name>.<port_name>". This allows to specify a port that should be connected to the pad by default after reset. An error is raised if the specified default_port is not actually connectable to the pad (not in the same mux group).
- Added new CLI commands to customize internal Mako templates.
- Add optional user_attr key to padframe specification format. This allows users to tag pads, ports etc. with additional metadata and potentially use it within the custom templates.
- Add 'generate constraints' command to auto-generate SDC constraints for set_case_analysis of pad_mux config registers.
- Add new CLI generate command to render custom templates to support completely customized output formats
- Add more documentation on usage and structure of generated padrick output

1.8.9.2 Changed

- Manifest version was increased to 2.
- Renamed *mux_group* key to *mux_groups* which now accepts a list of string instead of a single string.
- Updated sample config files to be compatible with new manifest format.
- Switched to reg_interface version 3.1 and updated internal reggen version.
- Config file types of various fields to support expression language (makes fields like *default_static_value* incompatible with YAML integers)

1.8.9.3 Fixed

- Fixed bug in toplevel padfram struct generation

1.8.10 0.1.0 - 2021-03-30

Very first *alpha* release of Padrick with support for RTL Generation and Driver Generation.

1.9 padrick

1.9.1 padrick package

1.9.1.1 Subpackages

padrick.Generators package

Subpackages

padrick.Generators.ConstraintsGenerator package

Subpackages

padrick.Generators.ConstraintsGenerator.Templates package

Module contents

Submodules

padrick.Generators.ConstraintsGenerator.ConstraintsGenerator module

padrick.Generators.ConstraintsGenerator.ConstraintsGenerator.**generate_constraints**(*templates:* ConstraintsTemplates, *padframe:* Padframe, *constraints_spec:* ConstraintsSpec, *dir:* *Path*, *header_text:* *str*, ***extra_template_kwargs*)

padrick.Generators.ConstraintsGenerator.ConstraintsSpec module

exception padrick.Generators.ConstraintsGenerator.ConstraintsSpec.**ConstraintsGenException**

Bases: Exception

class padrick.Generators.ConstraintsGenerator.ConstraintsSpec.**ConstraintsMode**(*, *name:* *str*, *pad_domain:* *str*, *pad_mode:* *List*[ConstraintsPadMode])

Bases: BaseModel

classmethod expand_multi_pad_modes(*pad_configs:* *List*[ConstraintsPadMode])

link_with_pad_domain(*padframe:* Padframe)

name: str

pad_domain: str

pad_mode: List[*ConstraintsPadMode*]

class padrick.Generators.ConstraintsGenerator.ConstraintsSpec.**ConstraintsPadMode**(*, *pad_inst: Union[TemplatedIdentifierTy PadIn- stance]*, *port_sel: Op- tional[Union[TemplatedStrin Tu- ple[PortGroup, Port]]]* = *None*, *pad_cfg: Op- tional[Mapping[Union[PadS str]*, *Union[ConstrainedStrValue, int]]]* = *None*, *multiple: Con- strained- IntValue* = *1*)

Bases: BaseModel

expand_pad_mode() → List[*ConstraintsPadMode*]

link_with_pad_domain(*pad_domain:* PadDomain)

multiple: ConstrainedIntValue

pad_cfg: Optional[Mapping[Union[*PadSignal*, str], Union[ConstrainedStrValue, int]]]

pad_inst: Union[*TemplatedIdentifierType*, *PadInstance*]

port_sel: Optional[Union[*TemplatedStringType*, Tuple[*PortGroup*, *Port*]]]

classmethod validate_pad_cfg_expression_valid(*pad_cfg: Mapping[Union[*PadSignal*, str], str]*)

class padrick.Generators.ConstraintsGenerator.ConstraintsSpec.**ConstraintsSpec**(*, *mani- fest_version: Constrained- IntValue*, *modes: List[*ConstraintsMode*]*)

Bases: BaseModel

classmethod check_manifest_version(*version*)

Verifies that the configuration file has the right version number for the current version of padrick.

link_with_pad_domain(*padframe:* Padframe)

manifest_version: ConstrainedIntValue

modes: List[*ConstraintsMode*]

Module contents

padrick.Generators.DocGenerator package

Subpackages

padrick.Generators.DocGenerator.Templates package

Module contents

Submodules

padrick.Generators.DocGenerator.DocGenerator module

exception padrick.Generators.DocGenerator.DocGenerator.**DocGenException**

Bases: Exception

padrick.Generators.DocGenerator.DocGenerator.**generate_padlist**(*padframe:* Padframe, *dir:* *Path*)

Module contents

padrick.Generators.DriverGenerator package

Subpackages

padrick.Generators.DriverGenerator.Templates package

Module contents

Submodules

padrick.Generators.DriverGenerator.DriverGenerator module

exception padrick.Generators.DriverGenerator.DriverGenerator.**DriverGenException**

Bases: Exception

padrick.Generators.DriverGenerator.DriverGenerator.**generate_driver**(*templates:* DriverTemplates, *padframe:* Padframe, *dir:* *Path*, *header_text:* *str*, ***extra_template_kwargs*)

Module contents

padrick.Generators.FuseSoCGenerator package

Submodules

padrick.Generators.FuseSoCGenerator.FuseSoCGenerator module

padrick.Generators.FuseSoCGenerator.FuseSoCGenerator.**generate_core**(*config_file_path: Path*)

Parses the config file supplied by FuseSoC to generate a valid FuseSoC core config file.

padrick.Generators.FuseSoCGenerator.FuseSoCGeneratorConfigFileModel module

class padrick.Generators.FuseSoCGenerator.FuseSoCGeneratorConfigFileModel.**ConfigFileModel**(*, *files_root: Path*, *gapi: typing_extensions.Li*, *vlnv: str*, *parameters: ConfigFileParameters*, ***extra_data: Any*)

Bases: BaseModel

A pydantic data validation class to validate the generator config files supplied by FuseSoC to padrick's 'fusesoc_gen' command.

class Config

Bases: object

extra = 'allow'

files_root: Path

gapi: typing_extensions.Literal[1.0]

parameters: *ConfigFileParameters*

vlnv: str

class padrick.Generators.FuseSoCGenerator.FuseSoCGeneratorConfigFileModel.**ConfigFileParameters**(*, *generator_setting* *Optional[Path]* = *None*, *padframe_ma* *Path*, *generate_steps:* *List[Union* padrick.Ge Custom-GenerateStep]]], *padrick_cm* *Optional[str]* = *None*)

Bases: BaseModel

generate_steps: List[Union[*padrick.Generators.FuseSoCGenerator.FuseSoCGeneratorConfigFileModel.RTLGenerateStep*, *padrick.Generators.FuseSoCGenerator.FuseSoCGeneratorConfigFileModel.CustomGenerateStep*][Union[*RTLGenerateStep*, *CustomGenerateStep*]]]

generator_settings: Optional[Path]

padframe_manifest: Path

padrick_cmd: Optional[str]

class padrick.Generators.FuseSoCGenerator.FuseSoCGeneratorConfigFileModel.**ConstraintsGenerateStep**(*, *kind: typing_ex*

Bases: *GenerateStep*

kind: typing_extensions.Literal[constraints]

class padrick.Generators.FuseSoCGenerator.FuseSoCGeneratorConfigFileModel.**CustomGenerateStep**(*, *kind: typing_extensions.Literal[custom]*, *template_file: Path*, *output_filename: Path*)

Bases: *GenerateStep*

kind: typing_extensions.Literal[custom]

output_filename: Path

template_file: Path

class padrick.Generators.FuseSoCGenerator.FuseSoCGeneratorConfigFileModel.**DriverGenerateStep**(*, *kind: typing_extensions.Literal[driver]*)

Bases: *GenerateStep*

kind: typing_extensions.Literal[driver]

class padrick.Generators.FuseSoCGenerator.FuseSoCGeneratorConfigFileModel.**GenerateStep**(*, *kind: str*)

Bases: BaseModel

kind: str

class padrick.Generators.FuseSoCGenerator.FuseSoCGeneratorConfigFileModel.**GeneratorKind**(*value*)

Bases: str, Enum

An enumeration.

constraints = 'constraints'

custom = 'custom'

driver = 'driver'

padlist = 'padlist'

rtl = 'rtl'

class padrick.Generators.FuseSoCGenerator.FuseSoCGeneratorConfigFileModel.**PadlistGenerateStep**(*, *kind: typing_extensions.Literal[padlist]*)

Bases: *GenerateStep*

kind: typing_extensions.Literal[padlist]

class padrick.Generators.FuseSoCGenerator.FuseSoCGeneratorConfigFileModel.**RTLGenerateStep**(*, *kind: typing_extensions.Li*

Bases: *GenerateStep*

kind: typing_extensions.Literal[rtl]

Module contents

padrick.Generators.RTLGenerator package

Subpackages

padrick.Generators.RTLGenerator.Templates package

Module contents

Submodules

padrick.Generators.RTLGenerator.RTLGenerator module

exception padrick.Generators.RTLGenerator.RTLGenerator.**RTLGenException**

Bases: Exception

padrick.Generators.RTLGenerator.RTLGenerator.**generate_rtl**(*templates:* RTLTemplates, *padframe:* Padframe, *dir:* *Path*, *header_text:* *str*, *vlnv=None*, ***extra_template_kwargs*)

Module contents

Submodules

padrick.Generators.CLIGeneratorCommands module

padrick.Generators.GeneratorSettings module

class padrick.Generators.GeneratorSettings.**ConstraintsTemplates**(*, *case_analysis:* PadrickTemplate = *PadrickTemplate(name='Set Case Analysis statements for padmultiplexer', target_file_name='{padframe.name}_mode_{constrain tem-plate=TemplatePackageResource(package='padrick re-source='set_case_analysis.sdc.mako'), skip_generation=False)*)

Bases: BaseModel

class padrick.Generators.GeneratorSettings.**DocTemplates**

Bases: BaseModel

class padrick.Generators.GeneratorSettings.**DriverTemplates**(*, *regfile_hjson:* PadrickTemplate = *PadrickTemplate(name='Register File Specification for {pad_domain.name}', target_file_name='{padframe.name}_{pad_domain.name}_re template=TemplatePackageResource(package='padrick.Gene resource='regfile.hjson.mako'), skip_generation=False)*, *driver_header:* PadrickTemplate = *PadrickTemplate(name='Driver header file', target_file_name='{padframe.name}.h', template=TemplatePackageResource(package='padrick.Gene resource='driver.h.mako'), skip_generation=False)*, *driver_source:* PadrickTemplate = *PadrickTemplate(name='Driver implementation file', target_file_name='{padframe.name}.c', template=TemplatePackageResource(package='padrick.Gene resource='driver.c.mako'), skip_generation=False)*)

Bases: BaseModel

class padrick.Generators.GeneratorSettings.**GeneratorSettings**(*, *manifest_version: ConstrainedIntValue = 3*, *rtl_templates:* RTLTemplates = *RTLTemplates(toplevel_sv_package=PadrickTemplate(name='S package', target_file_name='pkg_{padframe.name}.sv', template=TemplatePackageResource(package='padrick.Ge resource='pkg_padframe.sv.mako'), skip_generation=False), pad_domain_top=PadrickTemplate(name='Paddomain module {pad_domain.name}', target_file_name='{padframe.name}_{pad_domain.name} template=TemplatePackageResource(package='padrick.Ge resource='pad_domain.sv.mako'), skip_generation=False), pad_inst_module=PadrickTemplate(name='Pad instantiation module {pad_domain.name}', target_file_name='{padframe.name}_{pad_domain.name} template=TemplatePackageResource(package='padrick.Ge resource='pads.sv.mako'), skip_generation=False), internal_pkg=PadrickTemplate(name='Internal package for {pad_domain.name}', target_file_name='pkg_internal_{padframe.name}_{pad_d template=TemplatePackageResource(package='padrick.Ge resource='pkg_pad_domain_internals.sv.mako'), skip_generation=False), pad_mux_module=PadrickTemplate(name='Pad Multiplexer for {pad_domain.name}', target_file_name='{padframe.name}_{pad_domain.name} template=TemplatePackageResource(package='padrick.Ge resource='pad_multiplexer.sv.mako'), skip_generation=False), regfile_hjson=PadrickTemplate(name='Register File Specification for {pad_domain.name}', target_file_name='{padframe.name}_{pad_domain.name} template=TemplatePackageResource(package='padrick.Ge resource='regfile.hjson.mako'), skip_generation=False), toplevel_module=PadrickTemplate(name='Padframe Top Module', target_file_name='{padframe.name}.sv', template=TemplatePackageResource(package='padrick.Ge resource='padframe.sv.mako'), skip_generation=False), assign_header_file=PadrickTemplate(name='Padframe*

Bases: BaseModel

classmethod check_manifest_version(*version*)

Verifies that the configuration file has the right version number for the current version of padrick.

manifest_version: ConstrainedIntValue

class padrick.Generators.GeneratorSettings.**RTLTemplates**(*, *toplevel_sv_package*: PadrickTemplate = *PadrickTemplate(name='SV package', target_file_name='pkg_{padframe.name}.sv', template=TemplatePackageResource(package='padrick.Generato resource='pkg_padframe.sv.mako'), skip_generation=False)*, *pad_domain_top*: PadrickTemplate = *PadrickTemplate(name='Paddomain module {pad_domain.name}', target_file_name='{padframe.name}_{pad_domain.name}.sv', template=TemplatePackageResource(package='padrick.Generato resource='pad_domain.sv.mako'), skip_generation=False)*, *pad_inst_module*: PadrickTemplate = *PadrickTemplate(name='Pad instantiation module {pad_domain.name}', target_file_name='{padframe.name}_{pad_domain.name}_pads. template=TemplatePackageResource(package='padrick.Generato resource='pads.sv.mako'), skip_generation=False)*, *internal_pkg*: PadrickTemplate = *PadrickTemplate(name='Internal package for {pad_domain.name}', target_file_name='pkg_internal_{padframe.name}_{pad_domain template=TemplatePackageResource(package='padrick.Generato resource='pkg_pad_domain_internals.sv.mako'), skip_generation=False)*, *pad_mux_module*: PadrickTemplate = *PadrickTemplate(name='Pad Multiplexer for {pad_domain.name}', target_file_name='{padframe.name}_{pad_domain.name}_muxe template=TemplatePackageResource(package='padrick.Generato resource='pad_multiplexer.sv.mako'), skip_generation=False)*, *regfile_hjson*: PadrickTemplate = *PadrickTemplate(name='Register File Specification for {pad_domain.name}', target_file_name='{padframe.name}_{pad_domain.name}_regs.h template=TemplatePackageResource(package='padrick.Generato resource='regfile.hjson.mako'), skip_generation=False)*, *toplevel_module*: PadrickTemplate = *PadrickTemplate(name='Padframe Top Module', target_file_name='{padframe.name}.sv', template=TemplatePackageResource(package='padrick.Generato resource='padframe.sv.mako'), skip_generation=False)*, *assign_header_file*: PadrickTemplate = *PadrickTemplate(name='Padframe assignment header file', target_file_name='assign.svh', tem-

Bases: BaseModel

padrick.Generators.PadrickTemplate module

class padrick.Generators.PadrickTemplate.**PadrickTemplate**(*, *name: str*, *target_file_name: str*, *template: Union[TemplatePackageResource, Path]*, *skip_generation: bool = False*)

Bases: BaseModel

name: str

render(*output_dir: Path*, *logger: Logger*, *padframe:* Padframe, *debug_render=False*, ***kwargs*)

target_file_name: str

template: Union[*TemplatePackageResource*, Path]

class padrick.Generators.PadrickTemplate.**TemplatePackageResource**(*package*, *resource*)

Bases: tuple

property package

Alias for field number 0

property resource

Alias for field number 1

exception padrick.Generators.PadrickTemplate.**TemplateRenderException**

Bases: Exception

Module contents

padrick.Model package

Submodules

padrick.Model.CommonValidators module

padrick.Model.CommonValidators.**check_sv_literal**(*literal: str*) → str

padrick.Model.Constants module

padrick.Model.PadDomain module

class padrick.Model.PadDomain.**PadDomain**(**args*, *name: ConstrainedStrValue*, *description: Optional[str] = None*, *pad_types: ConstrainedListValue[PadType]*, *pad_list: ConstrainedListValue[PadInstance]*, *port_groups: List[PortGroup] = []*, *user_attr: Optional[Dict[str, Union[str, int, bool]]] = None*)

Bases: BaseModel

A pad_domain contains the configuration about one collection of pads and ports that can connected with each other.

classmethod check_each_pad_instance_name_is_unique(*pads: List[PadInstance]*)

classmethod check_padsignal_with_same_name_have_same_size_and_direction(*values*)

classmethod check_port_group_names_are_unique(*port_groups: List[PortGroup]*)

classmethod check_static_connection_signals_are_not_bidirectional(*v*)

description: Optional[str]

property dynamic_pad_signals: List[*Signal*]

property dynamic_pad_signals_pad2soc

property dynamic_pad_signals_soc2pad: List[*Signal*]

classmethod error_on_empty_port_groups_but_existing_dynamic_pads(*values*)

classmethod error_on_nonempty_port_groups_but_without_any_dynamic_pads(*values*)

classmethod expand_multi_pads(*pads: List[PadInstance]*)

classmethod expand_multi_port_groups(*port_groups: List[PortGroup]*)

get_dynamic_pad_signals_for_mux_group(*mux_group: str*) → List[*Signal*]

get_dynamic_pad_signals_pad2soc_for_mux_group(*mux_group: str*)

get_dynamic_pad_signals_soc2pad_for_mux_group(*mux_group: str*) → List[*Signal*]

get_dynamic_pads_in_mux_groups(*mux_groups: Set[str]*) → List[*Port*]

get_ports_in_mux_groups(*mux_groups: Set[str]*) → List[*Port*]

name: ConstrainedStrValue

classmethod normalize_pad_mux_groups(*pads: List[PadInstance]*)

classmethod normalize_port_mux_groups(*port_groups: List[PortGroup]*, *values*)

classmethod override_port_mux_group(*port_groups: List[PortGroup]*, *values*)

property override_signals: List[*Signal*]

pad_list: ConstrainedListValue[*PadInstance*]

property pad_mux_group_sets: List[Set[str]]

pad_types: ConstrainedListValue[*PadType*]

port_groups: List[*PortGroup*]

property port_mux_group_sets: List[Set[str]]

property static_connection_signals: List[*Signal*]

property static_connection_signals_pad2soc: List[*Signal*]

property static_connection_signals_soc2pad: List[*Signal*]

user_attr: Optional[Dict[str, Union[str, int, bool]]]

classmethod validate_and_link_default_ports(*values*)

classmethod warn_about_orphan_pads_and_ports(*values*)

padrick.Model.PadInstance module

class padrick.Model.PadInstance.**PadInstance**(**, name: ~padrick.Model.TemplatedIdentifier.TemplatedIdentifierType, description: ~typing.Optional[~padrick.Model.TemplatedString.TemplatedStringType] = None, multiple: ~padrick.Model.PadInstance.ConstrainedIntValue = 1, pad_type: ~typing.Union[~padrick.Model.PadInstance.ConstrainedStrValue, ~padrick.Model.PadType.PadType], is_static: bool = False, mux_groups: ~types.ConstrainedSetValue[~padrick.Model.TemplatedIdentifier.TemplatedId = {all, self}, connections: ~typing.Optional[~typing.Mapping[~typing.Union[~padrick.Model.PadSignal.Pad str], ~typing.Optional[~padrick.Model.SignalExpressionType.SignalExpressionType]]] = None, default_port: ~typing.Optional[~typing.Union[~typing.Mapping[~typing.Union[typing_extensio ~padrick.Model.TemplatedIdentifier.TemplatedIdentifierType], ~padrick.Model.TemplatedPortIdentifier.TemplatedPortIdentifierType], ~padrick.Model.TemplatedPortIdentifier.TemplatedPortIdentifierType, ~typing.Tuple[~padrick.Model.PortGroup.PortGroup, ~padrick.Model.Port.Port]]] = None, user_attr: ~typing.Optional[~padrick.Model.UserAttrs.UserAttrs] = None*)

Bases: BaseModel

class Config

Bases: object

extra = 'forbid'

underscore_attrs_are_private = True

validate_assignment = True

connections: Optional[Mapping[Union[*PadSignal*, str], Optional[*SignalExpressionType*]]]

default_port: Port]]]

description: Optional[*TemplatedStringType*]

property dynamic_pad_signals

property dynamic_pad_signals_pad2soc

property dynamic_pad_signals_soc2pad

expand_padinstance() → List[*PadInstance*]

is_static: bool

property landing_pads

classmethod link_and_validate_connections(*v: Mapping[str,* SignalExpressionType*]*, *values*)

classmethod lookup_pad_type(*v: Union[*PadType*, str]*) → *PadType*

multiple: ConstrainedIntValue

property mux_group_name

mux_groups: ConstrainedSetValue[*TemplatedIdentifierType*]

classmethod mux_groups_must_not_contain_uppercase_letters(*mux_group:* TemplatedIdentifierType)

name: *TemplatedIdentifierType*

classmethod no_connections_for_pad_signal_of_kind_pad(*values*)

property override_signals: List[*Signal*]

pad_type: Union[ConstrainedStrValue, *PadType*]

property static_connection_signals: List[*Signal*]

Returns all static connection signals used for the given pad. Returns:

property static_pad_signal_connections

property static_pad_signals

user_attr: Optional[*UserAttrs*]

padrick.Model.PadSignal module

class padrick.Model.PadSignal.ConnectionType(*value*)

Bases: str, Enum

An enumeration.

dynamic = 'dynamic'

static = 'static'

class padrick.Model.PadSignal.**PadSignal**(*direction=None*, **values*, *name: ~padrick.Model.TemplatedIdentifier.TemplatedIdentifierType*, *size: ~padrick.Model.PadSignal.ConstrainedIntValue = 1*, *description: ~typing.Optional[str] = None*, *kind: ~padrick.Model.PadSignal.PadSignalKind*, *conn_type: ~typing.Optional[~padrick.Model.PadSignal.ConnectionType] = None*, *and_override_signal: ~padrick.Model.SignalExpressionType.SignalExpressionType =*, *or_override_signal: ~padrick.Model.SignalExpressionType.SignalExpressionType =*, *default_reset_value: ~typing.Optional[int] = None*, *default_static_value: ~typing.Optional[~padrick.Model.SignalExpressionType.SignalExpressionType] = None*, *user_attr: ~typing.Optional[~padrick.Model.UserAttrs.UserAttrs] = None*)

Bases: *Signal*

class Config

Bases: object

extra = 'forbid'

and_override_signal: *SignalExpressionType*

conn_type: Optional[*ConnectionType*]

default_reset_value: Optional[int]

default_static_value: Optional[*SignalExpressionType*]

description: Optional[str]

property direction

kind: *PadSignalKind*

classmethod must_contain_conn_type_unsless_kind_pad(*values*)

classmethod must_contain_default_values_if_kind_input(*values*)

classmethod must_not_contain_default_value_if_landing_pad(*values*)

or_override_signal: *SignalExpressionType*

property static_signals

user_attr: Optional[*UserAttrs*]

classmethod validate_output_pad(*values*)

class padrick.Model.PadSignal.**PadSignalKind**(*value*)

Bases: str, Enum

An enumeration.

input = 'input'

output = 'output'

pad = 'pad'

class padrick.Model.PadSignal.**Signal**(*direction=None*, **values*, *name:* TemplatedIdentifierType, *size: ConstrainedIntValue = 1*)

Bases: BaseModel

property direction

name: *TemplatedIdentifierType*

size: ConstrainedIntValue

class padrick.Model.PadSignal.**SignalDirection**(*value*)

Bases: str, Enum

An enumeration.

bidir = 'bidir'

pads2soc = 'pads2soc'

soc2pads = 'soc2pads'

padrick.Model.PadType module

class padrick.Model.PadType.**PadType**(**args*, *name: ConstrainedStrValue*, *description: Optional[str] = None*, *template: str*, *pad_signals: List[*PadSignal*] = []*, *user_attr: Optional[*UserAttrs*] = None*)

Bases: BaseModel

class Config

Bases: object

extra = 'forbid'

underscore_attrs_are_private = True

classmethod check_unique_padtype_name(*v*)

classmethod check_valid_mako_template(*v*)

description: Optional[str]

get_pad_signal(*name: str*) → *PadSignal*

classmethod must_contain_at_least_one_landing_pad(*v: List[*PadSignal*]*) → List[*PadSignal*]

name: ConstrainedStrValue

pad_signals: List[*PadSignal*]

template: str

user_attr: Optional[*UserAttrs*]

padrick.Model.Padframe module

class padrick.Model.Padframe.**Padframe**(*, *manifest_version: int*, *name: ConstrainedStrValue*, *description: Optional[str] = None*, *pad_domains: ConstrainedListValue[PadDomain]*, *user_attr: Optional[UserAttrs] = None*)

Bases: `BaseModel`

Padframe class that represents the padframe configuration parsed from the configuration file.

manifest_version

The manifest version used by the parsed configuration file.

Type
int

name

Name of the pad_frame module.

Type
str

description

An optional short description of the padframes.

Type
str

pad_domains

A list of PadDomains within this padframe.

Type
List[*PadDomain*

class Config

Bases: `object`

```
json_encoders = {<class 'mako.template.Template'>:  <function
Padframe.Config.<lambda>>, <class
'padrick.Model.SignalExpressionType.SignalExpressionType'>:  <function
Padframe.Config.<lambda>>, <class 'padrick.Model.PadSignal.PadSignal'>:
<function Padframe.Config.<lambda>>, <class 'padrick.Model.PadSignal.Signal'>:
<function Padframe.Config.<lambda>>}
```

```
title = 'Padframe Config'
```

```
underscore_attrs_are_private = True
```

classmethod check_manifest_version(*version*)

Verifies that the configuration file has the right version number for the current version of padrick.

```
description:  Optional[str]
```

```
manifest_version:  int
```

```
name:  ConstrainedStrValue
```

```
pad_domains:  ConstrainedListValue[PadDomain]
```

```
user_attr:  Optional[UserAttrs]
```

padrick.Model.ParseContext module

class padrick.Model.ParseContext.**ParseContext**

Bases: object

find_pad_signal_instances(*name: str*) → List[*Signal*]

find_pad_type(*name: str*) → Optional[*PadType*]

register_pad_type(*pad_type:* PadType)

set_context(*ctx:* PadDomain)

padrick.Model.Port module

class padrick.Model.Port.**Port**(*, *name:* TemplatedIdentifierType, *description:* *Optional*[TemplatedStringType] = *None*, *connections:* *Optional*[*Mapping*[*Union*[Signal, *str*], *Optional*[SignalExpressionType]]] = *None*, *mux_groups: ConstrainedSetValue*[TemplatedIdentifierType] = *{all, self}*, *multiple: ConstrainedIntValue* = *1*, *user_attr:* *Optional*[UserAttrs] = *None*)

Bases: BaseModel

class Config

Bases: object

extra = 'forbid'

underscore_attrs_are_private = True

validate_assignment = True

connections: Optional[Mapping[Union[*Signal*, str], Optional[*SignalExpressionType*]]]

description: Optional[*TemplatedStringType*]

expand_port() → List[*Port*]

classmethod link_and_validate_connections(*v: Mapping*[*Union*[Signal, *str*], SignalExpressionType], *values*)

multiple: ConstrainedIntValue

property mux_group_name: str

mux_groups: ConstrainedSetValue[*TemplatedIdentifierType*]

classmethod mux_groups_must_not_contain_uppercase_letters(*mux_group:* TemplatedIdentifierType)

name: *TemplatedIdentifierType*

property port_signals: List[*Signal*]

The union of pad2chip and chip2pad port signals.

Type

Returns

property port_signals_chip2pad: List[*Signal*]

property port_signals_pad2chip: List[*Signal*]

user_attr: Optional[*UserAttrs*]

padrick.Model.PortGroup module

class padrick.Model.PortGroup.**PortGroup**(*, *name:* TemplatedIdentifierType, *description:* *Optional*[TemplatedStringType] = *None*, *mux_groups:* *Optional*[*ConstrainedSetValue*[TemplatedIdentifierType]] = *None*, *ports:* *List*[Port], *output_defaults:* *Union*[SignalExpressionType, *Mapping*[*Union*[Signal, *str*], *Optional*[SignalExpressionType]]] = *{}*, *multiple:* *ConstrainedIntValue* = *1*, *user_attr:* *Optional*[UserAttrs] = *None*)

Bases: BaseModel

class Config

Bases: object

extra = 'forbid'

underscore_attrs_are_private = True

classmethod check_all_pad2soc_ports_have_default(*values*)

classmethod check_pad2soc_ports_are_not_multiple_connected(*v*)

classmethod check_port_signals_are_not_bidirectional(*v*)

classmethod check_ports_are_unique(*ports*)

description: Optional[*TemplatedStringType*]

classmethod expand_default_value_for_connection_defaults(*output_defaults*, *values*)

classmethod expand_multi_ports(*ports*)

Expand ports with muliple>1 into individual port objects replacing the '<>' token in name, description and signalexpression with the array index.

expand_port_group() → List[*PortGroup*]

get_ports_in_mux_groups(*mux_groups: Set[str]*) → List[*Port*]

multiple: ConstrainedIntValue

mux_groups: Optional[ConstrainedSetValue[*TemplatedIdentifierType*]]

name: *TemplatedIdentifierType*

output_defaults: Union[*SignalExpressionType*, Mapping[Union[*Signal*, str], Optional[*SignalExpressionType*]]]

property port_signals

property port_signals_pads2soc

property port_signals_soc2pads

ports: List[*Port*]

user_attr: Optional[*UserAttrs*]

classmethod validate_and_link_output_defaults(*v: Mapping[str, SignalExpressionType]*, *values*)

Make sure the signals specified in connection_defaults are actually pad2chip port signals and make sure the associated expression is static.

padrick.Model.SignalExpressionType module

class padrick.Model.SignalExpressionType.**SignalExpressionType**(*expression: str*)

Bases: str

property ast

evaluate_template(*i*)

property expression: str

get_mapped_expr(*signal_name_mapping: Mapping[str, str]*) → *SignalExpressionType*

property is_const_expr

property is_empty

property is_single_signal

property signal_collection

classmethod validate(*v*)

class padrick.Model.SignalExpressionType.**SignalNameRemapTransformer**(*signal_name_mapping: Mapping[str, str]*)

Bases: Transformer

signal_name(*characters*)

padrick.Model.TemplatedIdentifier module

class padrick.Model.TemplatedIdentifier.**TemplatedIdentifierType**(*expression: str*)

Bases: str

property ast

evaluate_template(*i*)

property identifier: str

classmethod validate(*v*)

padrick.Model.TemplatedIdentifier.**parse_expression**(*expression: str*)

padrick.Model.TemplatedIndexGrammar module

class padrick.Model.TemplatedIndexGrammar.**TemplatedIdxEvaluator**(*i: int*)

Bases: Transformer

INDEX_VAR(*token*)

constant(*token*)

idx_expression(*left*, *operator*, *right*)

idx_template(*idx_expression*, *format_spec: Tree = None*)

number_to_base26(*value: int*) → List[int]

term(*left*, *operator*, *right*)

class padrick.Model.TemplatedIndexGrammar.**TemplatedIdxToStringTransformer**(*visit_tokens=True*)

Bases: Transformer

idx_template(*children*)

padrick.Model.TemplatedPortIdentifier module

class padrick.Model.TemplatedPortIdentifier.**TemplatedPortIdentifierType**(*expression: str*)

Bases: str

evaluate_template(*i*)

property identifier: str

classmethod validate(*v*)

padrick.Model.TemplatedString module

class padrick.Model.TemplatedString.**TemplatedStringType**(*expression: str*)

Bases: str

property ast

evaluate_template(*i*)

property identifier: str

classmethod validate(*v*)

padrick.Model.UserAttrs module

class padrick.Model.UserAttrs.**UserAttrs**(*, *__root__*: *Mapping*[TemplatedStringType, *Union*[UserAttrs, *int*, *bool*, TemplatedStringType]])

Bases: BaseModel

expand_user_attrs(*i: int*) → Dict[str, Union[str, int, bool]]

items()

keys()

values()

padrick.Model.Utilities module

padrick.Model.Utilities.**cached_property**(*func*)

padrick.Model.Utilities.**sort_pads**(*seq: Iterable*[PadInstance])

padrick.Model.Utilities.**sort_ports**(*seq: Iterable*[Port])

padrick.Model.Utilities.**sort_signals**(*seq: Iterable*[Signal])

Module contents

padrick.Utils package

Submodules

padrick.Utils.WorkingDir module

padrick.Utils.WorkingDir.**working_dir**(*path*)

A context manager to temporarily chang the working directory

Module contents

1.9.1.2 Submodules

1.9.1.3 padrick.CLIEntryPoint module

1.9.1.4 padrick.ConfigParser module

padrick.ConfigParser.**get_error_context**(*config_file: Path*, *line*, *column*, *context_before=4*, *context_after=4*)

padrick.ConfigParser.**get_file_location**(*config_data: CommentedMap*, *error_location: List*[*Union*[*str*, *int*]]) → Tuple[Tuple[int, int], Mapping]

padrick.ConfigParser.**get_human_readable_error_path**(*config_data: dict*, *error_location: List[Union[str, int]]*)

padrick.ConfigParser.**parse_config**(*cls: T*, *config_file: Path*, *include_base_dir: Optional[Path] = None*, *ignore_includes=False*) → Optional[T]

1.9.1.5 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

I

N

O

P

R

S

T

U

V

W