# Python productivity for Zynq (Pynq) Documentation

*Release 2.5*

**Xilinx**

**Feb 21, 2020**

# Contents

Xilinx® makes Zynq® and Zynq Ultrascale+™ devices, a class of programmable System on Chip (SoC) which integrates a multi-core processor (Dual-core ARM® Cortex®-A9 or Quad-core ARM® Cortex®-A53) and a Field Programmable Gate Array (FPGA) into a single integrated circuit. FPGA, or programmable logic, and microprocessors are complementary technologies for embedded systems. Each meets distinct requirements for embedded systems that the other cannot perform as well.

# Project Goals

The main goal of **PYNQ**, **Py**thon Productivity for Zy**nq**, is to make it easier for designers of embedded systems to exploit the unique benefits of Xilinx devices in their applications. Specifically, PYNQ enables architects, engineers and programmers who design embedded systems to use Zynq devices, without having to use ASIC-style design tools to design programmable logic circuits.

PYNQ achieves this goal in three ways:

- Programmable logic circuits are presented as hardware libraries called *overlays*. These overlays are analogous to software libraries. A software engineer can select the overlay that best matches their application. The overlay can be accessed through an application programming interface (API). Creating a new overlay still requires engineers with expertise in designing programmable logic circuits. The key difference however, is the *build once, re-use many times* paradigm. Overlays, like software libraries, are designed to be configurable and re-used as often as possible in many different applications.

**Note:** This is a familiar approach that borrows from best-practice in the software community. Every day, the Linux kernel is used by hundreds of thousands of embedded designers. The kernel is developed and maintained by fewer than one thousand, high-skilled, software architects and engineers. The extensive re-use of the work of a relatively small number of very talented engineers enables many more software engineers to work at higher levels of abstraction. Hardware libraries or *overlays* are inspired by the success of the Linux kernel model in abstracting so many of the details of low-level, hardware-dependent software.

- PYNQ uses Python for programming both the embedded processors and the overlays. Python is a "productivity-level" language. To date, C or C++ are the most common, embedded programming languages. In contrast, Python raises the level of programming abstraction and programmer productivity. These are not mutually-exclusive choices, however. PYNQ uses CPython which is written in C, and integrates thousands of C libraries and can be extended with optimized code written in C. Wherever practical, the more productive Python environment should be used, and whenever efficiency dictates, lower-level C code can be used.

- PYNQ is an open-source project that aims to work on any computing platform and operating system. This goal is achieved by adopting a web-based architecture, which is also browser agnostic. We incorporate the open-source Jupyter notebook infrastructure to run an Interactive Python (IPython) kernel and a web server directly on the ARM processor of the Zynq device. The web server brokers access to the kernel via a suite of browser-

based tools that provide a dashboard, bash terminal, code editors and Jupyter notebooks. The browser tools are implemented with a combination of JavaScript, HTML and CSS and run on any modern browser.

Summary

PYNQ is the first project to combine the following elements to simplify and improve APSoC design:

1. A high-level productivity language (Python in this case)

2. FPGA overlays with extensive APIs exposed as Python libraries

3. A web-based architecture served from the embedded processors, and

4. The Jupyter Notebook framework deployed in an embedded context

## 2.1 Getting Started

This guide will show you how to setup your development board and computer to get started using PYNQ. Any questions can be posted to the PYNQ support forum.

If you have one of the following boards, you can follow the quick start guide.

### 2.1.1 PYNQ-Z1 Setup Guide
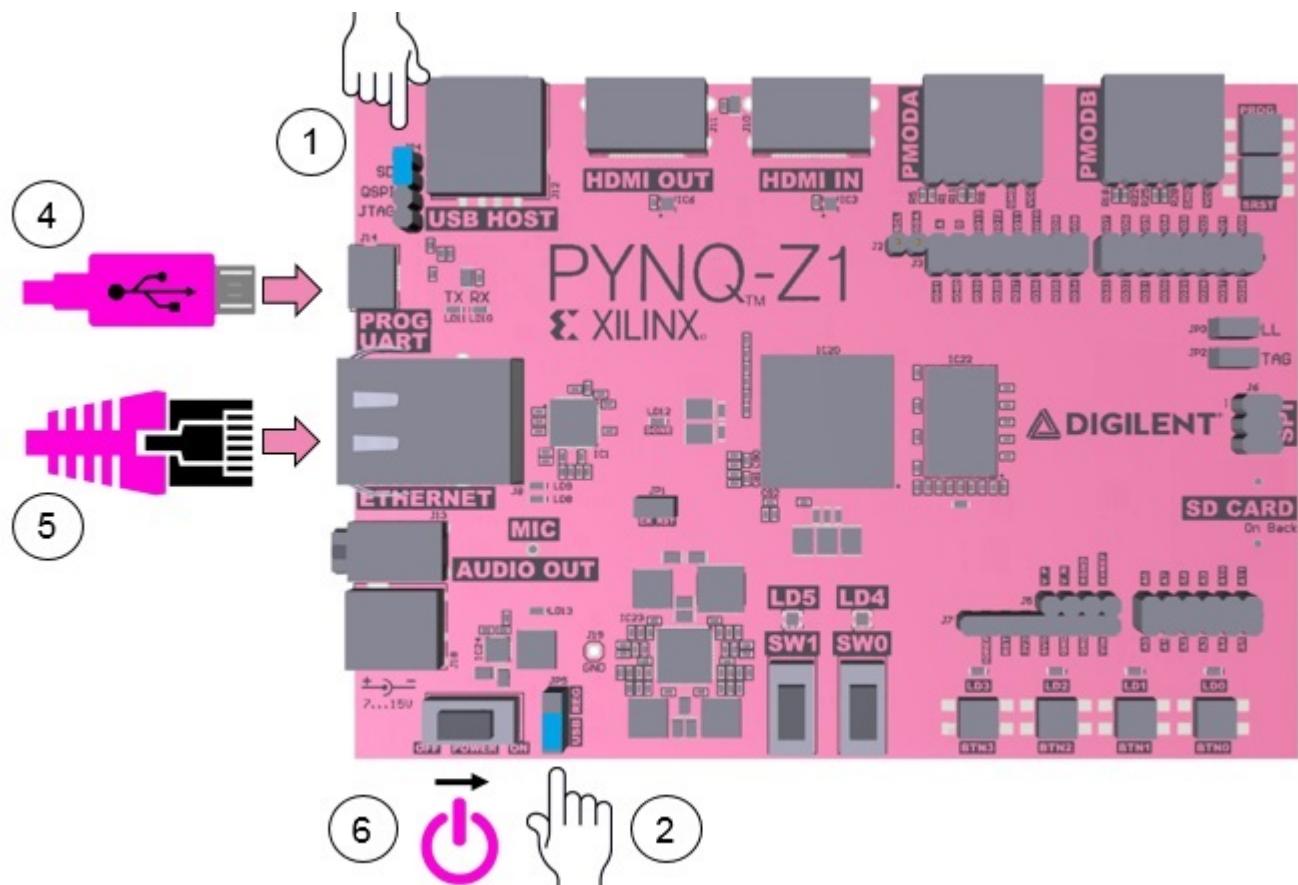
**Prerequisites for the Pynq-Z1**

- PYNQ-Z1 board

- Computer with compatible browser (Supported Browsers)

- Ethernet cable

- Micro USB cable

- Micro-SD card with preloaded image, or blank card (Minimum 8GB recommended)

### Getting Started Video

You can watch the getting started video guide, or follow the instructions in *Board Setup*.

### Board Setup



1. Set the **JP4 / Boot** jumper to the *SD* position by placing the jumper over the top two pins of JP4 as shown in the image. (This sets the board to boot from the Micro-SD card)

2. To power the PYNQ-Z1 from the micro USB cable, set the **JP5 / Power** jumper to the *USB* position. (You can also power the board from an external 12V power regulator by setting the jumper to *REG*.)

3. Insert the Micro SD card loaded with the PYNQ-Z1 image into the **Micro SD** card slot underneath the board.

4. Connect the USB cable to your PC/Laptop, and to the **PROG - UART / J14** MicroUSB port on the board

5. Connect the board to Ethernet by following the instructions below

6. Turn on the PYNQ-Z1 and check the boot sequence by following the instructions below

### Turning On the PYNQ-Z1

As indicated in step 6 of *Board Setup*, slide the power switch to the *ON* position to turn on the board. The **Red LD13** LED will come on immediately to confirm that the board has power. After a few seconds, the **Yellow/Green LD12 /**

**Done** LED will light up to show that the Zynq® device is operational.

After a minute you should see two **Blue LD4 & LD5** LEDs and four **Yellow/Green LD0-LD3** LEDs flash simultaneously. The **Blue LD4-LD5** LEDs will then turn on and off while the **Yellow/Green LD0-LD3** LEDs remain on. The system is now booted and ready for use.

### Network connection

Once your board is setup, you need to connect to it to start using Jupyter notebook.

### Ethernet

If available, you should connect your board to a network or router with Internet access. This will allow you to update your board and easily install new packages.

### Connect to a Computer

You will need to have an Ethernet port available on your computer, and you will need to have permissions to configure your network interface. With a direct connection, you will be able to use PYNQ, but unless you can bridge the Ethernet connection to the board to an Internet connection on your computer, your board will not have Internet access. You will be unable to update or load new packages without Internet access.

Connect directly to a computer (Static IP):

1. *Assign your computer a static IP address*

2. Connect the board to your computer's Ethernet port

3. Browse to http://192.168.2.99

### Connect to a Network Router

If you connect to a router, or a network with a DHCP server, your board will automatically get an IP address. You must make sure you have permission to connect a device to your network, otherwise the board may not connect properly.

Connect to a Router/Network (DHCP):

1. Connect the Ethernet port on your board to a router/switch

2. Connect your computer to Ethernet or WiFi on the router/switch

3. Browse to http://<board IP address>

4. Optional: *Change the Hostname*

5. Optional: *Configure Proxy Settings*

## 2.1.2 PYNQ-Z2 Setup Guide

### Prerequisites
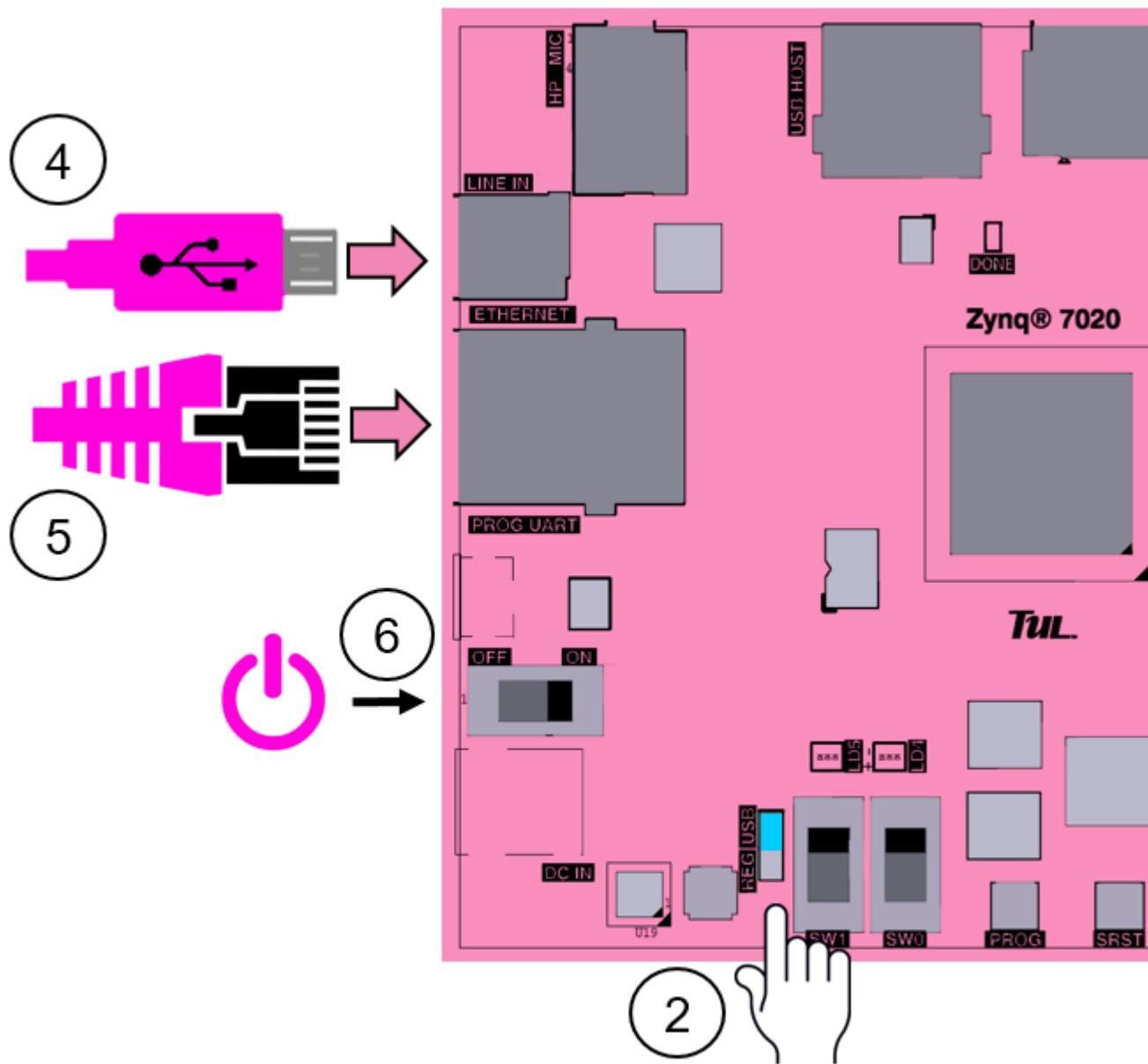
- PYNQ-Z2 board

- Computer with compatible browser (Supported Browsers)

- Ethernet cable

- Micro USB cable
- Micro-SD card with preloaded image, or blank card (Minimum 8GB recommended)

**Getting Started Video**

You can watch the getting started video guide, or follow the instructions in *Board Setup*.

**Board Setup**



1. Set the ** Boot** jumper to the *SD* position. (This sets the board to boot from the Micro-SD card)

2. To power the board from the micro USB cable, set the **Power** jumper to the *USB* position. (You can also power the board from an external 12V power regulator by setting the jumper to *REG*.)

3. Insert the Micro SD card loaded with the PYNQ-Z2 image into the **Micro SD** card slot underneath the board

4. Connect the USB cable to your PC/Laptop, and to the **PROG - UART** MicroUSB port on the board

5. Connect the Ethernet port by following the instructions below

6. Turn on the PYNQ-Z2 and check the boot sequence by following the instructions below

### Turning On the PYNQ-Z2

As indicated in step 6 of *Board Setup*, slide the power switch to the *ON* position to turn on the board. The **Red** LED will come on immediately to confirm that the board has power. After a few seconds, the **Yellow/Green / Done** LED will light up to show that the Zynq® device is operational.

After a minute you should see two **Blue ** LEDs and four **Yellow/Green** LEDs flash simultaneously. The **Blue** LEDs will then turn on and off while the **Yellow/Green** LEDs remain on. The system is now booted and ready for use.

### Network connection

Once your board is setup, you need to connect to it to start using Jupyter notebook.

### Ethernet

If available, you should connect your board to a network or router with Internet access. This will allow you to update your board and easily install new packages.

### Connect to a Computer

You will need to have an Ethernet port available on your computer, and you will need to have permissions to configure your network interface. With a direct connection, you will be able to use PYNQ, but unless you can bridge the Ethernet connection to the board to an Internet connection on your computer, your board will not have Internet access. You will be unable to update or load new packages without Internet access.

Connect directly to a computer (Static IP):

1. *Assign your computer a static IP address*

2. Connect the board to your computer's Ethernet port

3. Browse to http://192.168.2.99

### Connect to a Network Router

If you connect to a router, or a network with a DHCP server, your board will automatically get an IP address. You must make sure you have permission to connect a device to your network, otherwise the board may not connect properly.

Connect to a Router/Network (DHCP):

1. Connect the Ethernet port on your board to a router/switch

2. Connect your computer to Ethernet or WiFi on the router/switch

3. Browse to http://<board IP address>

4. Optional: *Change the Hostname*

5. Optional: *Configure Proxy Settings*

### 2.1.3 ZCU104 Setup Guide
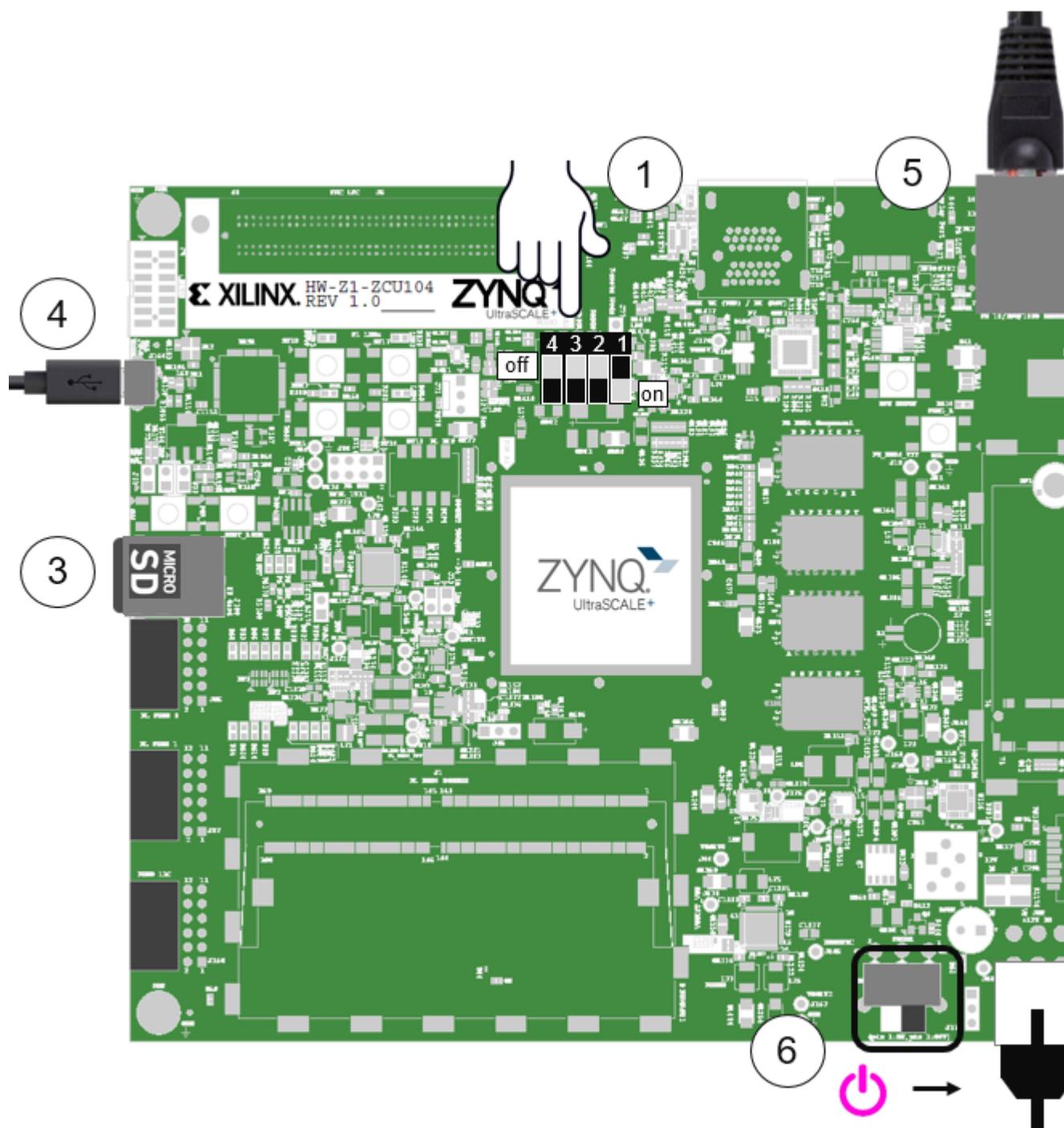
**Prerequisites**

- ZCU104 board
- Computer with compatible browser (Supported Browsers)
- Ethernet cable
- Micro USB cable (optional)
- Micro-SD card with preloaded image, or blank card (Minimum 8GB recommended)

**Getting Started Video**

You can watch the getting started video guide, or follow the instructions in *Board Setup*.

**Board Setup**



1. Set the **Boot** Dip Switches (SW6) to the following positions:

   (This sets the board to boot from the Micro-SD card)

   - Dip switch 1 (Mode 0): On (down position in diagram)
   - Dip switch 2 (Mode 1): Off (up position in diagram)
   - Dip switch 3 (Mode 2): Off (up)

- Dip switch 4 (Mode 3): Off (up)

2. Connect the 12V power cable. Note that the connector is keyed and can only be connected in one way.

3. Insert the Micro SD card loaded with the appropriate PYNQ image into the **MicroSD** card slot underneath the board

4. (Optional) Connect the USB cable to your PC/Laptop, and to the **USB JTAG UART** MicroUSB port on the board

5. Connect the Ethernet port by following the instructions below

6. Turn on the board and check the boot sequence by following the instructions below

### Turning On the ZCU104

As indicated in step 6, slide the power switch to the *ON* position to turn on the board. A **Red** LED and some additional yellow board LEDs will come on to confirm that the board has power. After a few seconds, the red LED will change to **Yellow**. This indicates that the bitstream has been downloaded and the system is booting.

### Network connection

Once your board is setup, you need to connect to it to start using Jupyter notebook.

### Ethernet

If available, you should connect your board to a network or router with Internet access. This will allow you to update your board and easily install new packages.

### Connect to a Computer

You will need to have an Ethernet port available on your computer, and you will need to have permissions to configure your network interface. With a direct connection, you will be able to use PYNQ, but unless you can bridge the Ethernet connection to the board to an Internet connection on your computer, your board will not have Internet access. You will be unable to update or load new packages without Internet access.

Connect directly to a computer (Static IP):

1. *Assign your computer a static IP address*

2. Connect the board to your computer's Ethernet port

3. Browse to http://192.168.2.99

### Connect to a Network Router

If you connect to a router, or a network with a DHCP server, your board will automatically get an IP address. You must make sure you have permission to connect a device to your network, otherwise the board may not connect properly.

Connect to a Router/Network (DHCP):

1. Connect the Ethernet port on your board to a router/switch

2. Connect your computer to Ethernet or WiFi on the router/switch

3. Browse to http://<board IP address>

4. Optional: *Change the Hostname*

5. Optional: *Configure Proxy Settings*

If you have another Zynq board see the following guide:

### 2.1.4 Using PYNQ with other Zynq boards

PYNQ is an open source software framework that supports Xilinx Zynq devices. To use PYNQ, a PYNQ image and suitable Zynq development board is required.

Currently two boards are officially supported by the PYNQ project (Pynq-Z1 from Digilent and Pynq-Z2 from TUL). This means that pre-compiled images are available for download for these boards that include example overlays and example Jupyter notebooks. All source files to rebuild the PYNQ image for these boards are available in the PYNQ GitHub. However, it is possible to use PYNQ with other Zynq development boards.

As PYNQ is a framework, there are a number of different components that make up PYNQ; Python libraries to control the Programmable Logic, Jupyter Notebook interface, PYNQ overlays and IP, pre-installed packages. There is a set of board requirements to use the full array of PYNQ features. This includes a network interface, USB port and UART. However, if a developer does not intend to use all aspects of PYNQ, only a subset of board features are required. For example, a design could be developed on a board fully supported by PYNQ, and deployed on a different production board. If Jupyter is only used for development, and not deployment, a network connection may not be required for the production board. For deployment, PYNQ designs and applications can be run on a minimal setup consisting of a Zynq device, boot source, and the minimum memory required to run an OS and the custom application.

The rest of this guide will assume a Zynq board will be used for development that contains all the recommended features for PYNQ development. This can be an off-the-shelf board, or a custom board you developed yourself.

#### Board recommendations for development

- Any Zynq/Zynq Ultrascale+ device (including single-core)
- >=512 MB DRAM
- SD Card (>=8GB) or other bootable source
- Network connection; Ethernet or WiFi
- UART
- USB

The network connection can be an Ethernet connection or a WiFi connection. USB WiFi, and boards with WiFi chips connected directly to the Zynq - E.g. SDIO or other interfaces can be used. Linux also has a feature to allow a USB port to be used as an Ethernet Gadget, allowing an Ethernet network connection to a PC over a USB cable.

UART is not essential, but can be useful to debug OS related issues. For example, UART can be used to check the network configuration and IP address of the board.

A USB port is not essential, but is useful if USB peripherals may be used with the board.

You will need a PYNQ image for your board. See the *PYNQ image* guide for details.

### 2.1.5 PYNQ image

**Pre-compiled images**

Pre-compiled images for supported boards can be found via the PYNQ boards page.

If you already have a MicroSD card preloaded with a PYNQ image for your board, you don't need to rewrite it unless you want to restore or update your image to a new version of PYNQ.

To write a PYNQ image, see the instructions below for *MicroSD Card Setup*.

**Other boards**

To use PYNQ with other Zynq boards, a PYNQ image is required.

If a PYNQ image is not already available for your board, you will need to build it yourself. You can do this by following the *PYNQ SD Card* guide.

You will need to setup and boot your board yourself, and setup a network connection to your computer to start using Jupyter. Once you do this, you can return to the *Connecting to Jupyter Notebook* instructions.

**MicroSD Card Setup**

To make your own PYNQ Micro-SD card:

1. Download the appropriate PYNQ image for your board

2. Unzip the image

3. Write the image to a blank Micro SD card (minimum 8GB recommended)

For detailed instructions on writing the SD card using different operating systems, see *Writing the SD Card Image*.
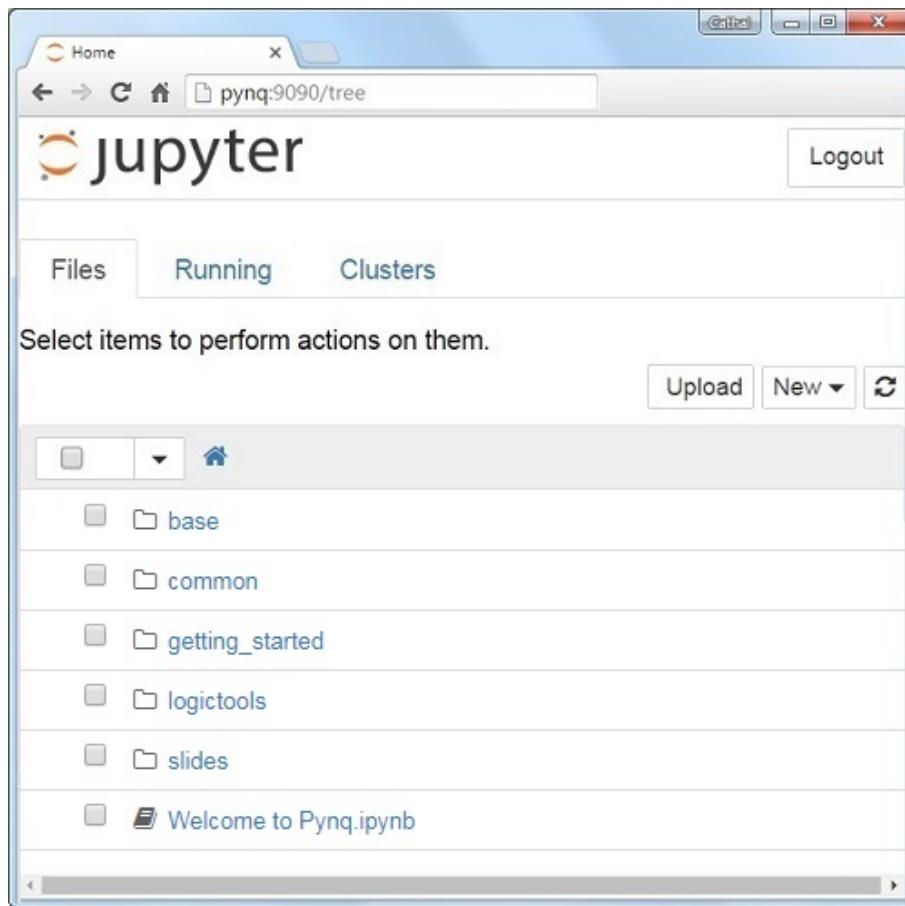
## 2.1.6 Connecting to Jupyter Notebook

Once your board is setup, to connect to Jupyter Notebooks open a web browser and navigate to:

- http://192.168.2.99 If your board is connected to a computer via a static IP address

If your board is configured correctly you will be presented with a login screen. The username is **xilinx** and the password is also **xilinx**.

After logging in, you will see the following screen:
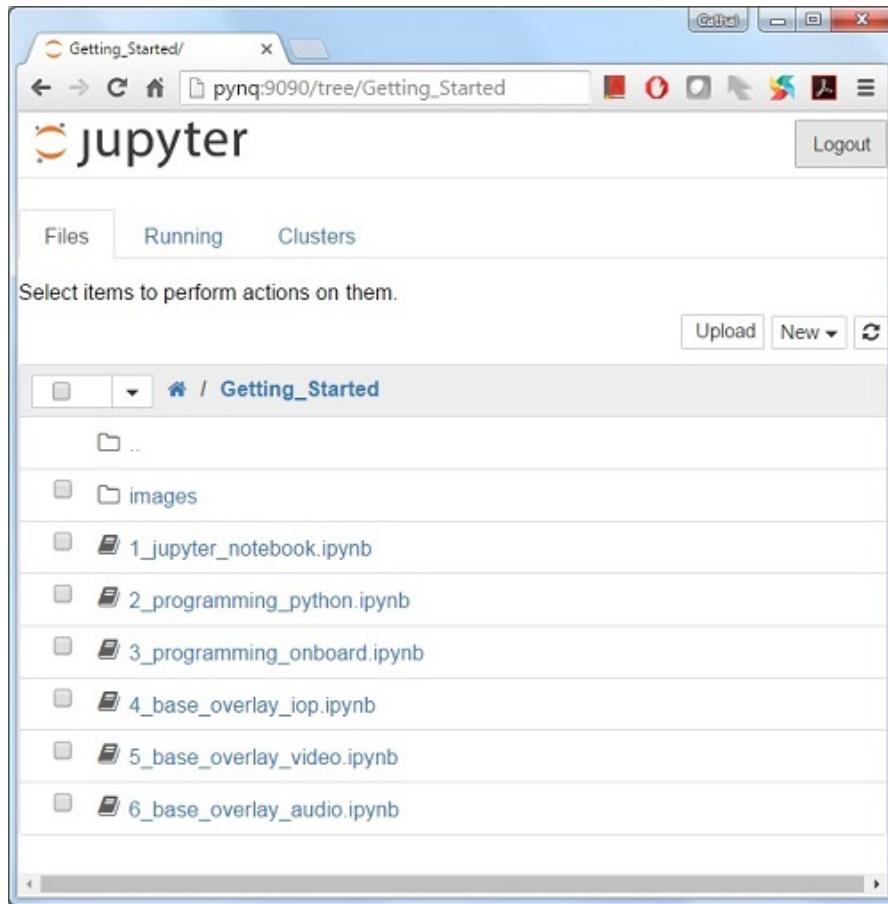
The default hostname is **pynq** and the default static IP address is **192.168.2.99**. If you changed the static IP of the board, you will need to change the address you browse to.

The first time you connect, it may take a few seconds for your computer to resolve the hostname/IP address.

### Example Notebooks

PYNQ uses the Jupyter Notebook environment to provide examples and documentation. Using your browser you can view and run the notebook documentation interactively.

The Getting_Started folder in the Jupyter home area includes some introductory Jupyter notebooks.

The example notebooks have been divided into categories

- common: examples that are not overlay specific

Depending on your board, and the PYNQ image you are using, other folders may be available with examples related to Overlays. E.g. The *base* directory will have examples related to the base overlay. If you install any additional overlays, a folder with example notebooks will usually be copied here.

When you open a notebook and make any changes, or execute cells, the notebook document will be modified. It is recommended that you "Save a copy" when you open a new notebook. If you want to restore the original versions, you can download all the example notebooks from GitHub.

### 2.1.7 Configuring PYNQ

#### Accessing Files on The Board
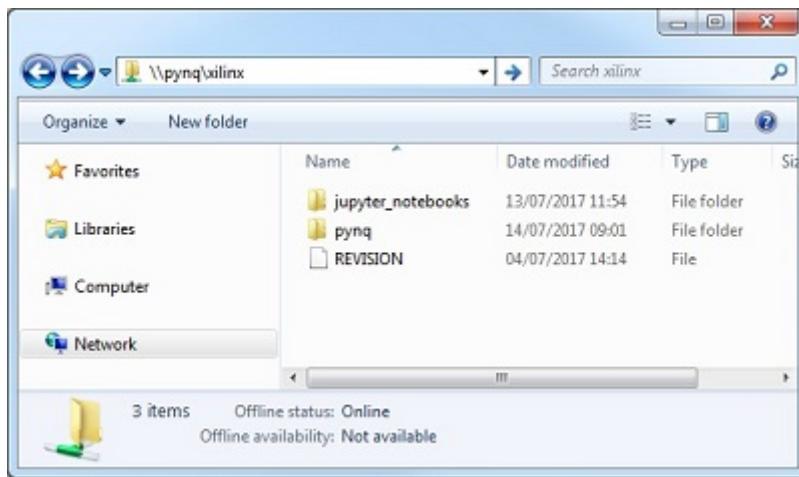
Samba, a file sharing service, is running on the board. This allows you to access the Pynq home area as a network drive, to transfer files to and from the board.

---

**Note:** In the examples below change the hostname or IP address to match your board settings.

---

To access the Pynq home area in Windows Explorer type one of the following in the navigation bar.

```
\\192.168.2.99\xilinx        # If connected to a Computer with a Static IP
```

When prompted, the username is **xilinx** and the password is **xilinx**. The following screen should appear:



To access the home area in Ubuntu, open a file broswer, click Go -> Enter Location and type one of the following in the box:
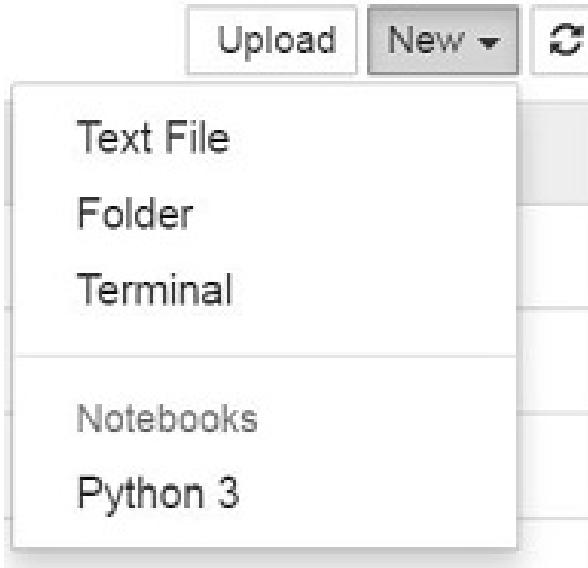
```
smb://192.168.2.99/xilinx        # If connected to a Computer with a Static IP
```

When prompted, the username is **xilinx** and the password is **xilinx**

### Change the Hostname

If you are on a network where other PYNQ boards may be connected, you should change your hostname immediately. This is a common requirement in a work or university environment. You can change the hostname from a terminal. You can use the USB cable to connect a terminal. A terminal is also available in the Jupyter environment and can be used from an internet browser.

To access the Jupyter terminal, in the Jupyter portal home area, select **New >> Terminal**.



This will open a terminal inside the browser as root.

Use the preloaded pynq_hostname.sh script to change your board's hostname.

```
pynq_hostname.sh <NEW HOSTNAME>
```

The board must be restarted for the changes to be applied.

```
shutdown -r now
```

Note that as you are logged in as root, sudo is not required. If you connect a terminal from the USB connection, you will be logged in as the *xilinx* user and sudo must be added to these commands.

When the board reboots, reconnect using the new hostname.

If you can't connect to your board, see the step below to open a terminal using the micro USB cable.

### Configure Proxy Settings

If your board is connected to a network that uses a proxy, you need to set the proxy variables on the board. Open a terminal as above and enter the following where you should replace "my_http_proxy:8080" and "my_https_proxy:8080" with your settings.

```
set http_proxy=my_http_proxy:8080
set https_proxy=my_https_proxy:8080
```

## 2.1.8 Troubleshooting

### Opening a USB Serial Terminal

If you can't access the terminal from Jupyter, you can connect the micro-USB cable from your computer to the board and open a terminal. You can use the terminal to check the network connection of the board. You will need to have terminal emulator software installed on your computer. PuTTY is one application that can be used, and is available for free on Windows. To open a terminal, you will need to know the COM port for the board.

On Windows, you can find this in the Windows *Device Manager* in the control panel.

1. Open the Device Manager, expand the *Ports* menu

2. Find the COM port for the *USB Serial Port*. e.g. COM5

3. Open PuTTY

Once PuTTY is open, enter the following settings:

4. Select serial

5. Enter the COM port number

6. Enter the serial terminal settings (below)

7. Click *Open*

Full terminal Settings:

- 115200 baud

- 8 data bits

- 1 stop bit

- No Parity

- No Flow Control

Hit *Enter* in the terminal window to make sure you can see the command prompt:

```
xilinx@pynq:/home/xilinx#
```

You can then run the same commands listed above to change the hostname, or configure a proxy.

You can also check the hostname of the board by running the *hostname* command:

```
hostname
```

You can also check the IP address of the board using *ifconfig*:

```
ifconfig
```

If you are having problems, please see the Troubleshooting section in *Frequently Asked Questions (FAQs)* or go the PYNQ support forum

## 2.2 Jupyter Notebooks

### 2.2.1 Acknowledgements

The material in this tutorial is specific to PYNQ. Wherever possible, however, it re-uses generic documentation describing Jupyter notebooks. In particular, we have re-used content from the following example notebooks:

1. What is the Jupyter Notebook?

2. Notebook Basics

3. Running Code

4. Markdown Cells

The original notebooks and further example notebooks are available at Jupyter documentation.

### 2.2.2 Introduction

If you are reading this documentation from the webpage, you should note that the webpage is a static html version of the notebook from which it was generated. If the PYNQ platform is available, you can open this notebook from the getting_started folder in the PYNQ Jupyter landing page.

The Jupyter Notebook is an **interactive computing environment** that enables users to author notebook documents that include:

- Live code

- Interactive widgets

- Plots

- Narrative text

- Equations

- Images

- Video

These documents provide a **complete and self-contained record of a computation** that can be converted to various formats and shared with others electronically, using version control systems (like git/GitHub) or nbviewer.jupyter.org.

### Components

The Jupyter Notebook combines three components:

- **The notebook web application**: An interactive web application for writing and running code interactively and authoring notebook documents.

- **Kernels**: Separate processes started by the notebook web application that runs users' code in a given language and returns output back to the notebook web application. The kernel also handles things like computations for interactive widgets, tab completion and introspection.

- **Notebook documents**: Self-contained documents that contain a representation of all content in the notebook web application, including inputs and outputs of the computations, narrative text, equations, images, and rich media representations of objects. Each notebook document has its own kernel.

### Notebook web application

The notebook web application enables users to:

- **Edit code in the browser**, with automatic syntax highlighting, indentation, and tab completion/introspection.

- **Run code from the browser**, with the results of computations attached to the code which generated them.

- See the results of computations with **rich media representations**, such as HTML, LaTeX, PNG, SVG, PDF, etc.

- Create and use **interactive JavaScript widgets**, which bind interactive user interface controls and visualizations to reactive kernel side computations.

- Author **narrative text** using the Markdown markup language.

- Build **hierarchical documents** that are organized into sections with different levels of headings.

- Include mathematical equations using **LaTeX syntax in Markdown**, which are rendered in-browser by Math-Jax.

### Kernels

The Notebook supports a range of different programming languages. For each notebook that a user opens, the web application starts a kernel that runs the code for that notebook. Each kernel is capable of running code in a single programming language. There are kernels available in the following languages:

- Python https://github.com/ipython/ipython

- Julia https://github.com/JuliaLang/IJulia.jl

- R https://github.com/takluyver/IRkernel

- Ruby https://github.com/minrk/iruby

- Haskell https://github.com/gibiansky/IHaskell

- Scala https://github.com/Bridgewater/scala-notebook

- node.js https://gist.github.com/Carreau/4279371

- Go https://github.com/takluyver/igo

PYNQ is written in Python, which is the default kernel for Jupyter Notebook, and the only kernel installed for Jupyter Notebook in the PYNQ distribution.

Kernels communicate with the notebook web application and web browser using a JSON over ZeroMQ/WebSockets message protocol that is described here. Most users don't need to know about these details, but its important to understand that kernels run on Zynq, while the web browser serves up an interface to that kernel.

### 2.2.3 Notebook Documents

Notebook documents contain the **inputs and outputs** of an interactive session as well as **narrative text** that accompanies the code but is not meant for execution. **Rich output** generated by running code, including HTML, images, video, and plots, is embedded in the notebook, which makes it a complete and self-contained record of a computation.

When you run the notebook web application on your computer, notebook documents are just **files** on your local filesystem with a **.ipynb** extension. This allows you to use familiar workflows for organizing your notebooks into folders and sharing them with others.

Notebooks consist of a **linear sequence of cells**. There are four basic cell types:

- **Code cells:** Input and output of live code that is run in the kernel

- **Markdown cells:** Narrative text with embedded LaTeX equations

- **Heading cells:** Deprecated. Headings are supported in Markdown cells

- **Raw cells:** Unformatted text that is included, without modification, when notebooks are converted to different formats using nbconvert

Internally, notebook documents are JSON data with binary values base64 encoded. This allows them to be **read and manipulated programmatically** by any programming language. Because JSON is a text format, notebook documents are version control friendly.

**Notebooks can be exported** to different static formats including HTML, reStructeredText, LaTeX, PDF, and slide shows (reveal.js) using Jupyter's `nbconvert` utility. Some of documentation for Pynq, including this page, was written in a Notebook and converted to html for hosting on the project's documentation website.
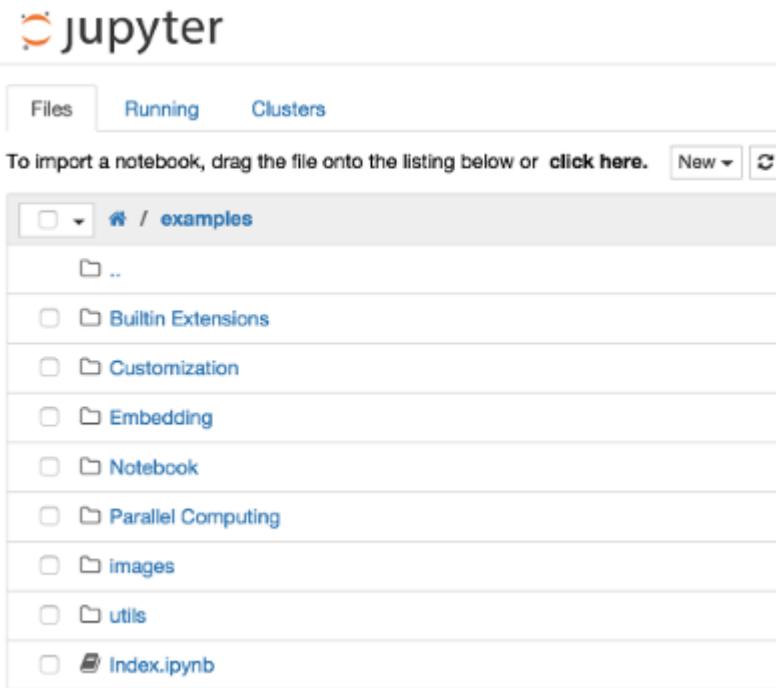
Furthermore, any notebook document available from a **public URL or on GitHub can be shared** via nbviewer. This service loads the notebook document from the URL and renders it as a static web page. The resulting web page may thus be shared with others **without their needing to install the Jupyter Notebook**.

GitHub also renders notebooks, so any Notebook added to GitHub can be viewed as intended.

### 2.2.4 Notebook Basics
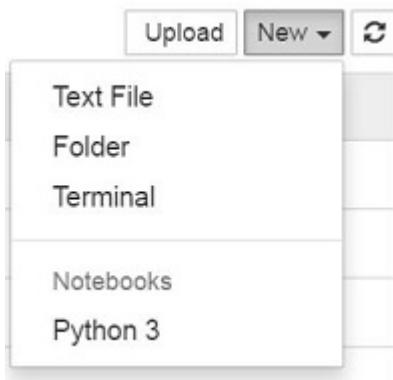
#### The Notebook dashboard

The Notebook server runs on the ARM® processor of the board. You can open the notebook dashboard by navigating to pynq:9090 when your board is connected to the network. The dashboard serves as a home page for notebooks. Its main purpose is to display the notebooks and files in the current directory. For example, here is a screenshot of the dashboard page for an example directory:
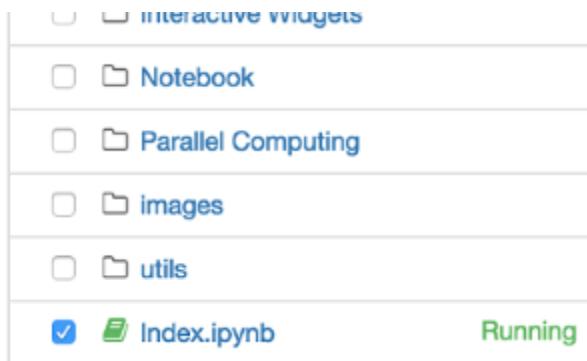
The top of the notebook list displays clickable breadcrumbs of the current directory. By clicking on these breadcrumbs or on sub-directories in the notebook list, you can navigate your filesystem.

To create a new notebook, click on the "New" button at the top of the list and select a kernel from the dropdown (as seen below).



Notebooks and files can be uploaded to the current directory by dragging a notebook file onto the notebook list or by the "click here" text above the list.

The notebook list shows green "Running" text and a green notebook icon next to running notebooks (as seen below). Notebooks remain running until you explicitly shut them down; closing the notebook's page is not sufficient.

To shutdown, delete, duplicate, or rename a notebook check the checkbox next to it and an array of controls will appear at the top of the notebook list (as seen below). You can also use the same operations on directories and files when applicable.



To see all of your running notebooks along with their directories, click on the "Running" tab:



This view provides a convenient way to track notebooks that you start as you navigate the file system in a long running notebook server.

## 2.2.5 Overview of the Notebook UI

If you create a new notebook or open an existing one, you will be taken to the notebook user interface (UI). This UI allows you to run code and author notebook documents interactively. The notebook UI has the following main areas:

- Menu
- Toolbar
- Notebook area and cells
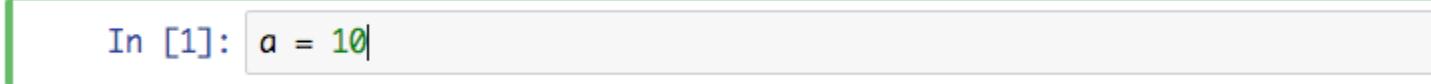
The notebook has an interactive tour of these elements that can be started in the "Help:User Interface Tour" menu item.

### Modal editor

The Jupyter Notebook has a modal user interface which means that the keyboard does different things depending on which mode the Notebook is in. There are two modes: edit mode and command mode.

### Edit mode

Edit mode is indicated by a green cell border and a prompt showing in the editor area:
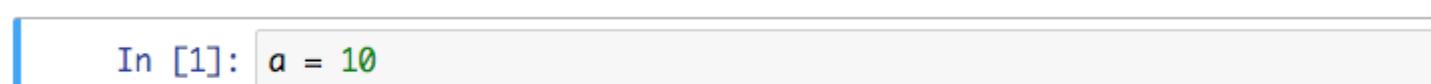


When a cell is in edit mode, you can type into the cell, like a normal text editor.

Enter edit mode by pressing `Enter` or using the mouse to click on a cell's editor area.

### Command mode

Command mode is indicated by a grey cell border with a blue left margin:



When you are in command mode, you are able to edit the notebook as a whole, but not type into individual cells. Most importantly, in command mode, the keyboard is mapped to a set of shortcuts that let you perform notebook and cell actions efficiently. For example, if you are in command mode and you press `c`, you will copy the current cell - no modifier is needed.

Don't try to type into a cell in command mode; unexpected things will happen!

Enter command mode by pressing `Esc` or using the mouse to click *outside* a cell's editor area.

### Mouse navigation

All navigation and actions in the Notebook are available using the mouse through the menubar and toolbar, both of which are above the main Notebook area:

Cells can be selected by clicking on them with the mouse. The currently selected cell gets a grey or green border depending on whether the notebook is in edit or command mode. If you click inside a cell's editor area, you will enter edit mode. If you click on the prompt or output area of a cell you will enter command mode.

If you are running this notebook in a live session on the board, try selecting different cells and going between edit and command mode. Try typing into a cell.

If you want to run the code in a cell, you would select it and click the `play` button in the toolbar, the "Cell:Run" menu item, or type Ctrl + Enter. Similarly, to copy a cell you would select it and click the `copy` button in the toolbar or the "Edit:Copy" menu item. Ctrl + C, V are also supported.

Markdown and heading cells have one other state that can be modified with the mouse. These cells can either be rendered or unrendered. When they are rendered, you will see a nice formatted representation of the cell's contents. When they are unrendered, you will see the raw text source of the cell. To render the selected cell with the mouse, and execute it. (Click the `play` button in the toolbar or the "Cell:Run" menu item, or type Ctrl + Enter. To unrender the selected cell, double click on the cell.

### Keyboard Navigation

There are two different sets of keyboard shortcuts: one set that is active in edit mode and another in command mode.

The most important keyboard shortcuts are `Enter`, which enters edit mode, and `Esc`, which enters command mode.

In edit mode, most of the keyboard is dedicated to typing into the cell's editor. Thus, in edit mode there are relatively few shortcuts. In command mode, the entire keyboard is available for shortcuts, so there are many more. The `Help->``Keyboard Shortcuts``` dialog lists the available shortcuts.

Some of the most useful shortcuts are:

1. Basic navigation: `enter`, `shift-enter`, `up/k`, `down/j`
2. Saving the notebook: `s`
3. Change Cell types: `y`, `m`, `1-6`, `t`
4. Cell creation: `a`, `b`
5. Cell editing: `x`, `c`, `v`, `d`, `z`
6. Kernel operations: `i`, `0` (press twice)

### 2.2.6 Running Code

First and foremost, the Jupyter Notebook is an interactive environment for writing and running code. The notebook is capable of running code in a wide range of languages. However, each notebook is associated with a single kernel. Pynq, and this notebook is associated with the IPython kernel, which runs Python code.

### Code cells allow you to enter and run code

Run a code cell using `Shift-Enter` or pressing the `play` button in the toolbar above. The button displays *run cell, select below* when you hover over it.

```
[1]: a = 10
```

```
[ ]: print(a)
```

There are two other keyboard shortcuts for running code:

- `Alt-Enter` runs the current cell and inserts a new one below.
- `Ctrl-Enter` run the current cell and enters command mode.

### Managing the Kernel

Code is run in a separate process called the Kernel. The Kernel can be interrupted or restarted. Try running the following cell and then hit the `stop` button in the toolbar above. The button displays *interrupt kernel* when you hover over it.

```
[ ]: import time
     time.sleep(10)
```

### Cell menu

The "Cell" menu has a number of menu items for running code in different ways. These includes:

- Run and Select Below
- Run and Insert Below
- Run All
- Run All Above
- Run All Below

### Restarting the kernels

The kernel maintains the state of a notebook's computations. You can reset this state by restarting the kernel. This is done from the menu bar, or by clicking on the corresponding button in the toolbar.

### sys.stdout

The stdout and stderr streams are displayed as text in the output area.

```
[ ]: print("Hello from Pynq!")
```

### Output is asynchronous

All output is displayed asynchronously as it is generated in the Kernel. If you execute the next cell, you will see the output one piece at a time, not all at the end.

```
[ ]: import time, sys
     for i in range(8):
         print(i)
         time.sleep(0.5)
```

**Large outputs**

To better handle large outputs, the output area can be collapsed. Run the following cell and then single- or double-click on the active area to the left of the output:

```python
for i in range(50):
    print(i)
```

### 2.2.7 Markdown

Text can be added to Jupyter Notebooks using Markdown cells. Markdown is a popular markup language that is a superset of HTML. Its specification can be found here:

http://daringfireball.net/projects/markdown/

**Markdown basics**

You can make text *italic* or **bold**.

You can build nested itemized or enumerated lists:

- One
    - Sublist
        * This
- Sublist - That - The other thing
- Two
- Sublist
- Three
- Sublist

Now another list:

1. Here we go
    1. Sublist
    2. Sublist
2. There we go
3. Now this

You can add horizontal rules:

---

Here is a blockquote:

> Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex. Complex is better than complicated. Flat is better than nested. Sparse is better than dense. Readability counts. Special cases aren't special enough to break the rules. Although practicality beats purity. Errors should never pass silently. Unless explicitly silenced. In the face of ambiguity, refuse the temptation to guess. There should be one– and preferably only one –obvious way to do it. Although that way may not be obvious at first unless you're Dutch. Now is better than never. Although never is often better than *right* now. If the implementation is hard to explain, it's a bad idea. If the implementation is easy to explain, it may be a good idea. Namespaces are one honking great idea – let's do more of those!

And shorthand for links:

Jupyter's website

### Headings

You can add headings by starting a line with one (or multiple) # followed by a space, as in the following example:

```
# Heading 1
# Heading 2
## Heading 2.1
## Heading 2.2
```

### Embedded code

You can embed code meant for illustration instead of execution in Python:

```python
def f(x):
    """a docstring"""
    return x**2
```

or other languages:

```c
if (i=0; i<n; i++) {
  printf("hello %d\n", i);
  x += 4;
}
```

### LaTeX equations

Courtesy of MathJax, you can include mathematical expressions inline or displayed on their own line.

Inline expressions can be added by surrounding the latex code with `$`:

```
Inline example: $e^{i\pi} + 1 = 0$
```

This renders as:

Inline example: $e^{i\pi} + 1 = 0$

Expressions displayed on their own line are surrounded by `$$`:

```
$$e^x=\sum_{i=0}^\infty \frac{1}{i!}x^i$$
```

This renders as:

$$e^x = \sum_{i=0}^{\infty} \frac{1}{i!}x^i$$

### GitHub flavored markdown

The Notebook webapp supports Github flavored markdown meaning that you can use triple backticks for code blocks:

```
<pre>
```python
print "Hello World"
```
</pre>

<pre>
```javascript
console.log("Hello World")
```
</pre>
```

Gives:

```python
print "Hello World"
```

```
console.log("Hello World")
```

And a table like this:

```
<pre>
```

| This | is    |
|------|------|
|   a  | table|

```
</pre>
```

A nice HTML Table:

| This | is    |
|------|-------|
| a    | table |

### General HTML

Because Markdown is a superset of HTML you can even add things like HTML tables:

Header 1

Header 2

row 1, cell 1

row 1, cell 2

row 2, cell 1

row 2, cell 2

### Local files

If you have local files in your Notebook directory, you can refer to these files in Markdown cells directly:

```
[subdirectory/]<filename>
```

### Security of local files

Note that the Jupyter notebook server also acts as a generic file server for files inside the same tree as your notebooks. Access is not granted outside the notebook folder so you have strict control over what files are visible, but for this reason it is highly recommended that you do not run the notebook server with a notebook directory at a high level in your filesystem (e.g. your home directory).

When you run the notebook in a password-protected manner, local file access is restricted to authenticated users unless read-only views are active. For more information, see Jupyter's documentation on running a notebook server.

## 2.3 Python Environment

We show here some examples of how to run Python on a Pynq platform. Python 3.6 is running exclusively on the ARM processor.

In the first example, which is based on calculating the factors and primes of integer numbers, give us a sense of the performance available when running on an ARM processor running Linux.

In the second set of examples, we leverage Python's `numpy` package and `asyncio` module to demonstrate how Python can communicate with programmable logic.

### 2.3.1 Factors and Primes Example

Code is provided in the cell below for a function to calculate factors and primes. It contains some sample functions to calculate the factors and primes of integers. We will use three functions from the `factors_and_primes` module to demonstrate Python programming.

```python
[1]: """Factors-and-primes functions.

Find factors or primes of integers, int ranges and int lists
and sets of integers with most factors in a given integer interval

"""


def factorize(n):
    """Calculate all factors of integer n.

    """
    factors = []
    if isinstance(n, int) and n > 0:
        if n == 1:
            factors.append(n)
            return factors
        else:
            for x in range(1, int(n**0.5)+1):
                if n % x == 0:
                    factors.append(x)
```

<div align="right">(continues on next page)</div>

```python
                    factors.append(n//x)
            return sorted(set(factors))
    else:
        print('factorize ONLY computes with one integer argument > 0')


def primes_between(interval_min, interval_max):
    """Find all primes in the interval.

    """
    primes = []
    if (isinstance(interval_min, int) and interval_min > 0 and
        isinstance(interval_max, int) and interval_max > interval_min):
        if interval_min == 1:
            primes = [1]
        for i in range(interval_min, interval_max):
            if len(factorize(i)) == 2:
                primes.append(i)
        return sorted(primes)
    else:
        print('primes_between ONLY computes over the specified range.')


def primes_in(integer_list):
    """Calculate all unique prime numbers.

    """
    primes = []
    try:
        for i in (integer_list):
            if len(factorize(i)) == 2:
                primes.append(i)
        return sorted(set(primes))
    except TypeError:
        print('primes_in ONLY computes over lists of integers.')


def get_ints_with_most_factors(interval_min, interval_max):
    """Finds the integers with the most factors.

    """
    max_no_of_factors = 1
    all_ints_with_most_factors = []

    # Find the lowest number with most factors between i_min and i_max
    if interval_check(interval_min, interval_max):
        for i in range(interval_min, interval_max):
            factors_of_i = factorize(i)
            no_of_factors = len(factors_of_i)
            if no_of_factors > max_no_of_factors:
                max_no_of_factors = no_of_factors
                results = (i, max_no_of_factors, factors_of_i,\
                            primes_in(factors_of_i))
        all_ints_with_most_factors.append(results)

        # Find any larger numbers with an equal number of factors
        for i in range(all_ints_with_most_factors[0][0]+1, interval_max):
```

```python
            factors_of_i = factorize(i)
            no_of_factors = len(factors_of_i)
            if no_of_factors == max_no_of_factors:
                results = (i, max_no_of_factors, factors_of_i, \
                            primes_in(factors_of_i))
                all_ints_with_most_factors.append(results)
        return all_ints_with_most_factors
    else:
        print_error_msg()


def interval_check(interval_min, interval_max):
    """Check type and range of integer interval.

    """
    if (isinstance(interval_min, int) and interval_min > 0 and
        isinstance(interval_max, int) and interval_max > interval_min):
        return True
    else:
        return False


def print_error_msg():
    """Print invalid integer interval error message.

    """
    print('ints_with_most_factors ONLY computes over integer intervals where'
            ' interval_min <= int_with_most_factors < interval_max and'
            ' interval_min >= 1')
```

Next we will call the factorize() function to calculate the factors of an integer.

```python
[2]: factorize(1066)
```

```
[2]: [1, 2, 13, 26, 41, 82, 533, 1066]
```

The primes_between() function can tell us how many prime numbers there are in an integer range. Let's try it for the interval 1 through 1066. We can also use one of Python's built-in methods len() to count them all.

```python
[3]: len(primes_between(1, 1066))
```

```
[3]: 180
```

Additionally, we can combine len() with another built-in method, sum(), to calculate the average of the 180 prime numbers.

```python
[4]: primes_1066 = primes_between(1, 1066)
     primes_1066_average = sum(primes_1066) / len(primes_1066)
     primes_1066_average
```

```
[4]: 486.2055555555556
```

This result makes sense intuitively because prime numbers are known to become less frequent for larger number intervals. These examples demonstrate how Python treats functions as first-class objects so that functions may be passed as parameters to other functions. This is a key property of functional programming and demonstrates the power of Python.

In the next code snippet, we can use list comprehensions (a 'Pythonic' form of the map-filter-reduce template) to 'mine' the factors of 1066 to find those factors that end in the digit '3'.

```
[5]: primes_1066_ends3 = [x for x in primes_between(1, 1066)
                            if str(x).endswith('3')]
     print('{}'.format(primes_1066_ends3))
```

```
[3, 13, 23, 43, 53, 73, 83, 103, 113, 163, 173, 193, 223, 233, 263, 283, 293, 313,
353, 373, 383, 433, 443, 463, 503, 523, 563, 593, 613, 643, 653, 673, 683, 733, 743,
773, 823, 853, 863, 883, 953, 983, 1013, 1033, 1063]
```

This code tells Python to first convert each prime between 1 and 1066 to a string and then to return those numbers whose string representation end with the number '3'. It uses the built-in str() and endswith() methods to test each prime for inclusion in the list.

And because we really want to know what fraction of the 180 primes of 1066 end in a '3', we can calculate . . .

```
[6]: len(primes_1066_ends3) / len(primes_1066)
```

```
[6]: 0.25
```

These examples demonstrate how Python is a modern, multi-paradigmatic language. More simply, it continually integrates the best features of other leading languages, including functional programming constructs. Consider how many lines of code you would need to implement the list comprehension above in C and you get an appreciation of the power of productivity-layer languages. Higher levels of programming abstraction really do result in higher programmer productivity!

### 2.3.2 Numpy Data Movement

Code in the cells below show a very simple data movement code snippet that can be used to share data with programmable logic. We leverage the Python numpy package to manipulate the buffer on the ARM processors and can then send a buffer pointer to programmable logic for sharing data.

We do not assume what programmable logic design is loaded, so here we only allocate the needed memory space and show that it can manipulated as a numpy array and contains a buffer pointer attribute. That pointer can then can be passed to programmable logic hardware.

```
[7]: import numpy as np
     import pynq

     def get_pynq_buffer(shape, dtype):
         """ Simple function to call PYNQ's memory allocator with numpy attributes

         """
         return pynq.allocate(shape, dtype)
```

With the simple wrapper above, we can get access to memory that can be shared by both numpy methods and programmable logic.

```
[8]: buffer = get_pynq_buffer(shape=(4,4), dtype=np.uint32)
     buffer
```

```
[8]: CMABuffer([[0, 0, 0, 0],
               [0, 0, 0, 0],
               [0, 0, 0, 0],
               [0, 0, 0, 0]], dtype=uint32)
```

To double-check we show that the buffer is indeed a numpy array.

```
[9]: isinstance(buffer,np.ndarray)
```

```
[9]: True
```

To send the buffer pointer to programmable logic, we use its physical address which is what programmable logic would need to communicate using this shared buffer.

```
[10]: pl_buffer_address = hex(buffer.physical_address)
      pl_buffer_address
```

```
[10]: '0x16846000'
```

In this short example, we showed a simple allocation of a numpy array that is now ready to be shared with programmable logic devices. With numpy arrays that are accessible to programmable logic, we can quickly manipulate and move data across software and hardware.

### 2.3.3 Asyncio Integration

PYNQ also leverages the Python asyncio module for communicating with programmable logic devices through events (namely interrupts).

A Python program running on PYNQ can use the asyncio library to manage multiple IO-bound tasks asynchronously, thereby avoiding any blocking caused by waiting for responses from slower IO subsystems. Instead, the program can continue to execute other tasks that are ready to run. When the previously-busy tasks are ready to resume, they will be executed in turn, and the cycle is repeated.

Again, since we won't assume what interrupt enabled devices are loaded on programmable logic, we will show an example here a software-only asyncio example that uses asyncio's sleep method.

```python
[11]: import asyncio
      import random
      import time

      # Coroutine
      async def wake_up(delay):
          '''A function that will yield to asyncio.sleep() for a few seconds
             and then resume, having preserved its state while suspended

          '''
          start_time = time.time()
          print(f'The time is: {time.strftime("%I:%M:%S")}')

          print(f"Suspending coroutine 'wake_up' at 'await` statement\n")
          await asyncio.sleep(delay)

          print(f"Resuming coroutine 'wake_up' from 'await` statement")
          end_time = time.time()
          sleep_time = end_time - start_time
          print(f"'wake-up' was suspended for precisely: {sleep_time} seconds")
```

With the wake_up function defined, we then can add a new task to the event loop.

```python
[12]: delay = random.randint(1,5)
      my_event_loop = asyncio.get_event_loop()

      try:
          print("Creating task for coroutine 'wake_up'\n")
```

---

```python
    wake_up_task = my_event_loop.create_task(wake_up(delay))
    my_event_loop.run_until_complete(wake_up_task)
except RuntimeError as err:
    print (f'{err}' +
        ' - restart the Jupyter kernel to re-run the event loop')
finally:
    my_event_loop.close()
```
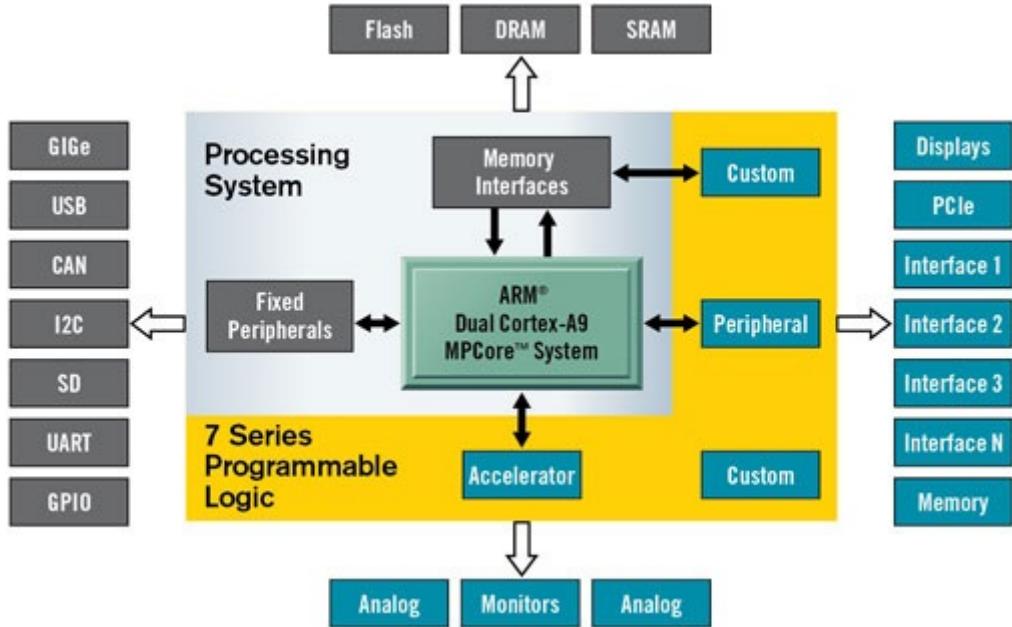
```
Creating task for coroutine 'wake_up'

The time is: 10:29:45
Suspending coroutine 'wake_up' at 'await` statement

Resuming coroutine 'wake_up' from 'await` statement
'wake-up' was suspended for precisely: 3.011084794998169 seconds
```

All the above examples show standard Python 3.6 running on the PYNQ platform. This entire notebook can be run on the PYNQ board - see the getting_started folder on the Jupyter landing page to rerun this notebook.

## 2.4 PYNQ Overlays

The Xilinx® Zynq® All Programmable device is an SOC based on a dual-core ARM® Cortex®-A9 processor (referred to as the *Processing System* or **PS**), integrated with FPGA fabric (referred to as *Programmable Logic* or **PL**). The *PS* subsystem includes a number of dedicated peripherals (memory controllers, USB, Uart, IIC, SPI etc) and can be extended with additional hardware IP in a *PL* Overlay.



Overlays, or hardware libraries, are programmable/configurable FPGA designs that extend the user application from the Processing System of the Zynq into the Programmable Logic. Overlays can be used to accelerate a software application, or to customize the hardware platform for a particular application.

For example, image processing is a typical application where the FPGAs can provide acceleration. A software programmer can use an overlay in a similar way to a software library to run some of the image processing functions (e.g. edge detect, thresholding etc.) on the FPGA fabric. Overlays can be loaded to the FPGA dynamically, as required,

just like a software library. In this example, separate image processing functions could be implemented in different overlays and loaded from Python on demand.

PYNQ provides a Python interface to allow overlays in the *PL* to be controlled from Python running in the *PS*. FPGA design is a specialized task which requires hardware engineering knowledge and expertise. PYNQ overlays are created by hardware designers, and wrapped with this PYNQ Python API. Software developers can then use the Python interface to program and control specialized hardware overlays without needing to design an overlay themselves. This is analogous to software libraries created by expert developers which are then used by many other software developers working at the application level.

## 2.4.1 Loading an Overlay

By default, an overlay (bitstream) called *base* is downloaded into the PL at boot time. The *base* overlay can be considered like a reference design for a board. New overlays can be installed or copied to the board and can be loaded into the PL as the system is running.

An overlay usually includes:

- A bitstream to configure the FPGA fabric
- A Vivado design Tcl file to determine the available IP
- Python API that exposes the IPs as attributes

The PYNQ `Overlay` class can be used to load an overlay. An overlay is instantiated by specifying the name of the bitstream file. Instantiating the Overlay also downloads the bitstream by default and parses the Tcl file.

```python
from pynq import Overlay
overlay = Overlay("base.bit")
```

For the base overlay, we can use the existing `BaseOverlay` class; this class exposes the IPs available on the bitstream as attributes of this class.

```
[1]: from pynq.overlays.base import BaseOverlay
     base_overlay = BaseOverlay("base.bit")
```

Once an overlay has been instantiated, the `help()` method can be used to discover what is in an overlay about. The *help* information can be used to interact with the overlay. Note that if you try the following code on your own board, you may see different results depending on the version of PYNQ you are using, and which board you have.

```
[2]: help(base_overlay)

Help on BaseOverlay in module pynq.overlays.base.base object:

class BaseOverlay(pynq.overlay.Overlay)
 |  The Base overlay for the Pynq-Z1
 |
 |  This overlay is designed to interact with all of the on board peripherals
 |  and external interfaces of the Pynq-Z1 board. It exposes the following
 |  attributes:
 |
 |  Attributes
 |  ----------
 |  iop_pmoda : IOP
 |      IO processor connected to the PMODA interface
 |  iop_pmodb : IOP
 |      IO processor connected to the PMODB interface
 |  iop_arduino : IOP
```

(continues on next page)

```
|        IO processor connected to the Arduino/ChipKit interface
| trace_pmoda : pynq.logictools.TraceAnalyzer
|        Trace analyzer block on PMODA interface, controlled by PS.
| trace_arduino : pynq.logictools.TraceAnalyzer
|        Trace analyzer block on Arduino interface, controlled by PS.
| leds : AxiGPIO
|        4-bit output GPIO for interacting with the green LEDs LD0-3
| buttons : AxiGPIO
|        4-bit input GPIO for interacting with the buttons BTN0-3
| switches : AxiGPIO
|        2-bit input GPIO for interacting with the switches SW0 and SW1
| rgbleds : [pynq.board.RGBLED]
|        Wrapper for GPIO for LD4 and LD5 multicolour LEDs
| video : pynq.lib.video.HDMIWrapper
|        HDMI input and output interfaces
| audio : pynq.lib.audio.Audio
|        Headphone jack and on-board microphone
|
| Method resolution order:
|     BaseOverlay
|     pynq.overlay.Overlay
|     pynq.pl.Bitstream
|     builtins.object
|
| Methods defined here:
|
| __init__(self, bitfile, **kwargs)
|     Return a new Overlay object.
|
|     An overlay instantiates a bitstream object as a member initially.
|
|     Parameters
|     ----------
|     bitfile_name : str
|         The bitstream name or absolute path as a string.
|     download : boolean or None
|         Whether the overlay should be downloaded. If None then the
|         overlay will be downloaded if it isn't already loaded.
|
|     Note
|     ----
|     This class requires a Vivado '.tcl' file to be next to bitstream file
|     with same name (e.g. base.bit and base.tcl).
|
| ----------------------------------------------------------------------
| Methods inherited from pynq.overlay.Overlay:
|
| __dir__(self)
|     __dir__() -> list
|     default dir() implementation
|
| __getattr__(self, key)
|     Overload of __getattr__ to return a driver for an IP or
|     hierarchy. Throws an `RuntimeError` if the overlay is not loaded.
|
| download(self)
|     The method to download a bitstream onto PL.
```

```
|
|       Note
|       ----
|       After the bitstream has been downloaded, the "timestamp" in PL will be
|       updated. In addition, all the dictionaries on PL will
|       be reset automatically.
|
|       Returns
|       -------
|       None
|
|   is_loaded(self)
|       This method checks whether a bitstream is loaded.
|
|       This method returns true if the loaded PL bitstream is same
|       as this Overlay's member bitstream.
|
|       Returns
|       -------
|       bool
|           True if bitstream is loaded.
|
|   load_ip_data(self, ip_name, data)
|       This method loads the data to the addressable IP.
|
|       Calls the method in the super class to load the data. This method can
|       be used to program the IP. For example, users can use this method to
|       load the program to the Microblaze processors on PL.
|
|       Note
|       ----
|       The data is assumed to be in binary format (.bin). The data name will
|       be stored as a state information in the IP dictionary.
|
|       Parameters
|       ----------
|       ip_name : str
|           The name of the addressable IP.
|       data : str
|           The absolute path of the data to be loaded.
|
|       Returns
|       -------
|       None
|
|   reset(self)
|       This function resets all the dictionaries kept in the overlay.
|
|       This function should be used with caution.
|
|       Returns
|       -------
|       None
|
|   ----------------------------------------------------------------------
|   Data descriptors inherited from pynq.pl.Bitstream:
|
```

```
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

This will give a list of the IP and methods available as part of the overlay.

From the `help()` print out above, it can be seen that in this case the overlay includes an `leds` instance, and from the report this is an AxiGPIO class:

```
"""
leds : AxiGPIO

    4-bit output GPIO for interacting with the green LEDs LD0-3
"""
```

Running `help()` on the leds object will provide more information about the object including details of its API.

```
[3]: help(base_overlay.leds)
```

```
Help on Channel in module pynq.lib.axigpio object:

class Channel(builtins.object)
 |  Class representing a single channel of the GPIO controller.
 |
 |  Wires are and bundles of wires can be accessed using array notation
 |  with the methods on the wires determined by the type of the channel::
 |
 |      input_channel[0].read()
 |      output_channel[1:3].on()
 |
 |  This class instantiated not used directly, instead accessed through
 |  the `AxiGPIO` classes attributes. This class exposes the wires
 |  connected to the channel as an array or elements. Slices of the
 |  array can be assigned simultaneously.
 |
 |  Methods defined here:
 |
 |  __getitem__(self, idx)
 |
 |  __init__(self, parent, channel)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __len__(self)
 |
 |  read(self)
 |      Read the state of the input pins
 |
 |  setdirection(self, direction)
 |      Set the direction of the channel
 |
 |      Must be one of AxiGPIO.{Input, Output, InOut} or the string
 |      'in', 'out', or 'inout'
 |
 |  setlength(self, length)
```

```
|        Set the number of wires connected to the channel
|
|  wait_for_interrupt_async(self)
|        Wait for the interrupt on the channel to be signalled
|
|        This is intended to be used by slices waiting for a particular
|        value but can be used in any situation to wait for a per-channel
|        interrupt.
|
|  write(self, val, mask)
|        Set the state of the output pins
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|        dictionary for instance variables (if defined)
|
|  __weakref__
|        list of weak references to the object (if defined)
|
|  trimask
|        Gets or sets the tri-state mask for an inout channel
```

The API can be used to control the object. For example, the following cell will turn on LD0 on the board.

```
[4]: base_overlay.leds[0].toggle()
```

Information about other IP can be found from the overlay instance in a similar way, as shown below.

```
[5]: help(base_overlay.video)

Help on HDMIWrapper in module pynq.lib.video object:

class HDMIWrapper(pynq.overlay.DefaultHierarchy)
 |  Hierarchy driver for the entire Pynq-Z1 video subsystem.
 |
 |  Exposes the input, output and video DMA as attributes. For most
 |  use cases the wrappers for the input and output pipelines are
 |  sufficient and the VDMA will not need to be used directly.
 |
 |  Attributes
 |  ----------
 |  hdmi_in : pynq.lib.video.HDMIIn
 |        The HDMI input pipeline
 |  hdmi_out : pynq.lib.video.HDMIOut
 |        The HDMI output pipeline
 |  axi_vdma : pynq.lib.video.AxiVDMA
 |        The video DMA.
 |
 |  Method resolution order:
 |        HDMIWrapper
 |        pynq.overlay.DefaultHierarchy
 |        pynq.overlay._IPMap
 |        builtins.object
 |
```

```
|  Methods defined here:
|
|  __init__(self, description)
|      Create a new _IPMap based on a hierarchical description.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  checkhierarchy(description)
|      Function to check if the driver matches a particular hierarchy
|
|      This function should be redefined in derived classes to return True
|      if the description matches what is expected by the driver. The default
|      implementation always returns False so that drivers that forget don't
|      get loaded for hierarchies they don't expect.
|
|  ----------------------------------------------------------------------
|  Methods inherited from pynq.overlay._IPMap:
|
|  __dir__(self)
|      __dir__() -> list
|      default dir() implementation
|
|  __getattr__(self, key)
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from pynq.overlay._IPMap:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

## 2.4.2 Partial Reconfiguration

From image v2.4, PYNQ supports partial bitstream reconfiguration. The partial bitstreams are managed by the *overlay* class. It is always recommended to use the *.hwh* file along with the *.bit* for the overlay class.

### Preparing the Files

There are many ways to prepare the bitstreams. Users can choose to follow the project flow or the software flow to implement a partial reconfiguration Vivado project. For more information, please refer to the documentation page on partial reconfiguration.

After each reconfiguration, the PL status will update to reflect the changes on the bitstream, so that new drivers can be assigned to the new blocks available in the bitstream. To achieve this, users have to provide the metadata file (*.hwh* file) along with each full / partial bitstream. The *.hwh* file is typically located at: *<project_name>/<design_name>.srcs/sources_1/bd/<design_name>/hw_handoff/*.

Keep in mind that each partial bitstream need a *.hwh* file.

### Loading Full Bitstream

It is straightforward to download a full bitstream. By default, the bitstream will be automatically downloaded onto the PL when users instantiate an overlay object.

```python
from pynq import Overlay
overlay = Overlay("full_bistream.bit')
```

To download the full bitstream again:

```python
overlay.download()
```

Note that no argument is provided if a full bitstream is to be downloaded.

Another thing to note, is that if the Vivado project is configured as a partial reconfiguration project, the *.hwh* file for the full bitstream will not contain any information inside a partial region, even if the full bitstream always has a default *Reconfiguration Module* (RM) implemented. Instead, the *.hwh* file only provides the information on the interfaces connecting to the partial region. So for the full bitstream, don't be surprised if you see an empty partial region in the *.hwh* file. The complete information on the partial regions are revealed by the *.hwh* files of the partial bitstreams, where each *.hwh* file reveals one possible internal organization of the partial region.

### Loading Partial Bitstream

Typically, the partial regions are hierarchies in the block design of the bitstream. In an *overlay* object, the hierarchical blocks are exposed as attributes of the object; users have to set the partial region for the overlay before they can reconfigure it. In the following example, let's assume there is a hierarchical block called *block_0* in the design.

```python
overlay.set_partial_region('block_0')
```

After the partial region is set, users can use the *download()* method for partial bitstreams. Note that an argument is now needed if a partial bitstream is to be downloaded.

```python
overlay.download('rm_0_partial.bit')
```

To load a different RM:

```python
overlay.download('rm_1_partial.bit')
```

## 2.4.3 PYNQ-Z1 Overlays

The PYNQ-Z1 board has the following features:

- Zynq XC7Z020-1CLG400C
- 512MB DDR3
- 1G Ethernet
- USB 2.0
- MicroSD
- Uart
- Microphone
- 3.5mm mono audio output jack

- 2x HDMI (can be used as input or output)

- 4 push-buttons

- 2 slide switches

- 4 LEDs

- 2 RGB LEDs

- 2x Pmod ports

- 1x Arduino header

For details on the PYNQ-Z1 board including PYNQ-Z1 reference manual and PYNQ-Z1 constraints file (xdc) see the PYNQ-Z1 webpage

The following overlays are include by default in the PYNQ image for the PYNQ-Z1 board:

### Base Overlay

The purpose of the base overlay design is to allow PYNQ to use peripherals on a board out-of-the-box. The design includes hardware IP to control peripherals on the target board, and connects these IP blocks to the Zynq PS. If a base overlay is available for a board, peripherals can be used from the Python environment immediately after the system boots.

Board peripherals typically include GPIO devices (LEDs, Switches, Buttons), Video, Audio, and other custom interfaces.

As the base overlay includes IP for the peripherals on a board, it can also be used as a reference design for creating new customized overlays.

In the case of general purpose interfaces, for example Pmod or Arduino headers, the base overlay may include a PYNQ MicroBlaze. A PYNQ MicroBlaze allows control of devices with different interfaces and protocols on the same port without requiring a change to the programmable logic design.

See *PYNQ Libraries* for more information on PYNQ MicroBlazes.

**PYNQ-Z1 Block Diagram**



The base overlay on PYNQ-Z1 includes the following hardware:

- HDMI (Input and Output)
- Microphone in
- Audio out
- User LEDs, Switches, Pushbuttons
- 2x Pmod PYNQ MicroBlaze
- Arduino PYNQ MicroBlaze
- 3x Trace Analyzer (PMODA, PMODB, ARDUINO)

**HDMI**

The PYNQ-Z1 has HDMI in and HDMI out ports. The HDMI interfaces are connected directly to PL pins. i.e. There is no external HDMI circuitry on the board. The HDMI interfaces are controlled by HDMI IP in the programmable logic.

The HDMI IP is connected to PS DRAM. Video can be streamed from the HDMI *in* to memory, and from memory to HDMI *out*. This allows processing of video data from python, or writing an image or Video stream from Python to the

HDMI out.

Note that while Jupyter notebook supports embedded video, video captured from the HDMI will be in raw format and would not be suitable for playback in a notebook without appropriate encoding.

For information on the physical HDMI interface ports, see the Digilent HDMI reference for the PYNQ-Z1 board

### HDMI In

The HDMI in IP can capture standard HDMI resolutions. After a HDMI source has been connected, and the HDMI controller for the IP is started, it will automatically detect the incoming data. The resolution can be read from the HDMI Python class, and the image data can be streamed to the PS DRAM.

### HDMI Out

The HDMI out IP supports the following resolutions:

- 640x480
- 800x600
- 1280x720 (720p)
- 1280x1024
- 1920x1080 (1080p)*

*While the Pynq-Z1 cannot meet the official HDMI specification for 1080p, some HDMI devices at this resolution may work.

Data can be streamed from the PS DRAM to the HDMI output. The HDMI Out controller contains framebuffers to allow for smooth display of video data.

See example video notebooks in the `<Jupyter Dashboard>/base/video` directory on the board.

### Microphone In

The PYNQ-Z1 board has an integrated microphone on the board and is connected directly to the Zynq PL pins, and does not have an external audio codec. The microphone generates audio data in PDM format.

For more information on the audio hardware, see the Digilent MIC in reference for the PYNQ-Z1 board

See example audio notebooks in the `<Jupyter Dashboard>/base/audio` directory on the board.

### Audio Out

The audio out IP is connected to a standard 3.5mm audio jack on the board. The audio output is PWM driven mono.

For more information on the audio hardware, see the Digilent Audio Out reference for the PYNQ-Z1 board

See example audio notebooks in the `<Jupyter Dashboard>/base/audio` directory on the board.

### User IO

The PYNQ-Z1 board includes two tri-color LEDs, 2 switches, 4 push buttons, and 4 individual LEDs. These IO are connected directly to Zynq PL pins. In the PYNQ-Z1 base overlay, these IO are routed to the PS GPIO, and can be controlled directly from Python.

### PYNQ MicroBlaze

PYNQ MicroBlazes are dedicated MicroBlaze soft-processor subsystems that allow peripherals with different IO standards to be connected to the system on demand. This allows a software programmer to use a wide range of peripherals with different interfaces and protocols. By using a PYNQ MicroBlaze, the same overlay can be used to support different peripheral without requiring a different overlay for each peripheral.

The PYNQ-Z1 has two types of PYNQ MicroBlaze: *Pmod* and *Arduino*, connecting to each type of corresponding interface. As the board has one Arduino header, and two Pmod ports, there is one instance of the Arduino, and two instances of the PMod PYNQ MicroBlaze in the base overlay.

Each physical interface has a different number of pins and can support different sets of peripherals. Each PYNQ MicroBlaze has the same core architecture, but can have different IP configurations to support the different sets of peripheral and interface pins.

PYNQ MicroBlaze block diagram and examples can be found in *MicroBlaze Subsystem*.

### Trace Analyzer

Trace analyzer blocks are connected to the interface pins for the two Pmod PYNQ MicroBlazes, and the Arduino PYNQ MicroBlaze. The trace analyzer can capture IO signals and stream the data to the PS DRAM for analysis in the Python environment.

Using the Python Wavedrom package, the signals from the trace analyzer can be displayed as waveforms in a Jupyter notebook.

On the base overlay, the trace analyzers are controlled by PS directly. In fact, on other overlays, the trace analyzers can also be controlled by PYNQ MicroBlaze.

See the example notebook in the `<Jupyter Dashboard>/base/trace` directory on the board.

### Python API

The Python API for the peripherals in the base overlay is covered in *PYNQ Libraries*. Example notebooks are also provided on the board to show how to use the base overlay.

### Rebuilding the Overlay

The project files for the overlays can be found here:

```
<PYNQ repository>/boards/<board>/base
```

### Linux

A Makefile is provided to rebuild the base overlay in Linux. The Makefile calls two tcl files. The first Tcl files compiles any HLS IP used in the design. The second Tcl builds the overlay.

To rebuild the overlay, source the Xilinx tools first. Then assuming PYNQ has been cloned:

```
cd <PYNQ repository>/boards/Pynq-Z1/base
make
```

### Windows

In Windows, the two Tcl files can be sourced in Vivado to rebuild the overlay. The Tcl files to rebuild the overlay can be sourced from the Vivado GUI, or from the Vivado Tcl Shell (command line).

To rebuild from the Vivado GUI, open Vivado. In the Vivado Tcl command line window change to the correct directory, and source the Tcl files as indicated below.

Assuming PYNQ has been cloned:

```
cd <PYNQ repository>/boards/Pynq-Z1/base
source ./build_base_ip.tcl
source ./base.tcl
```

To build from the command line, open the Vivado 2017.4 Tcl Shell, and run the following:
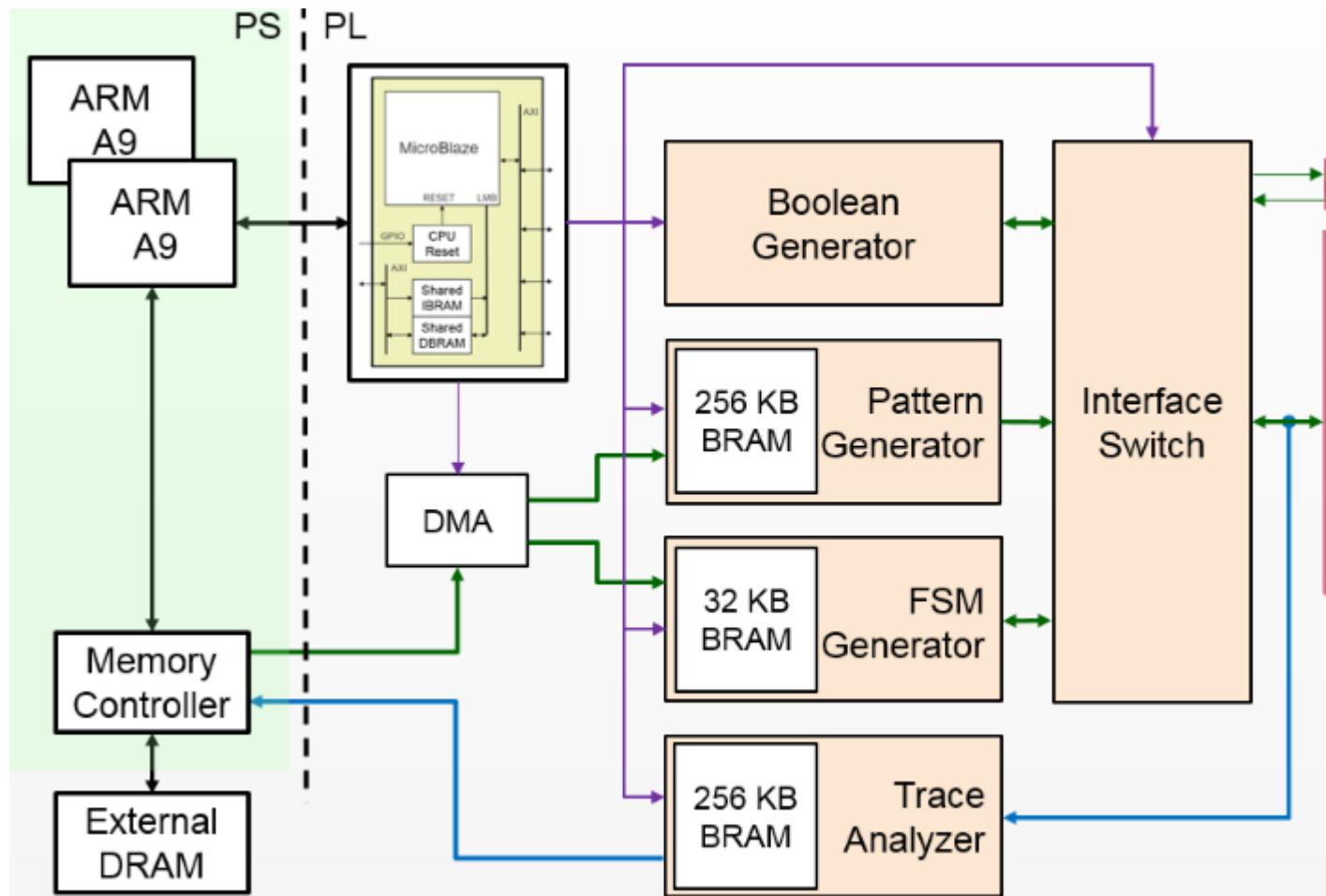
```
cd <PYNQ repository>/boards/Pynq-Z1/base
vivado -mode batch -source build_base_ip.tcl
vivado -mode batch -source base.tcl
```

Note that you must change to the overlay directory, as the tcl files has relative paths that will break if sourced from a different location.

### Logictools Overlay

The logictools overlay consists of programmable hardware blocks to connect to external digital logic circuits. Finite state machines, Boolean logic functions and digital patterns can be generated from Python. A programmable switch connects the inputs and outputs from the hardware blocks to external IO pins. The logictools overlay can also has a trace analyzer to capture data from the IO interface for analysis and debug.

**PYNQ-Z1 Block Diagram**



The logictools overlay on PYNQ-Z1 includes four main hardware blocks:

- Pattern Generator

- FSM Generator

- Boolean Generator

- Trace Analyzer

Each block is configured using a textual description specified in Python. No compilation of the configuration is required. This means a configuration can be loaded directly to the generator and run immediately.

**Pattern Generator**

The *Pattern Generator* can be configured to generate and stream arbitrary digital patterns to the external IO pins. The Pattern Generator can be used as a stimulus to test or control an external circuit.

### Finite State Machine (FSM) Generator

The *FSM Generator* can create a finite state machine from a Python description. The inputs and outputs and states of the FSM can be connected to external IO pins.

### Boolean Generator

The *Boolean Generator* can create independent combinatorial Boolean logic functions. The external IO pins are used as inputs and outputs to the Boolean functions.

### Trace Analyzer

The *Trace Analyzer* can capture IO signals and stream the data to the PS DRAM for analysis in the Python environment. The Trace Analyzer can be used standalone to capture external IO signals, or used in combination with the other three logictools functions to monitor data to and from the other blocks. E.g. the trace analyzer can be used with the pattern generator to verify the data sent to the external pins, or with the FSM to check the input, output or states to verify or debug a design.

### PYNQ MicroBlaze

A PYNQ MicroBlaze is used to control all the generators and analyzers. The PYNQ MicroBlaze subsystem on logictools overlay also manages contiguous memory buffers, configures the clock frequency, and keeps track of the generator status. For more information, please see *PYNQ Libraries*.

### Python API

The API for the logictools generators and trace analyzer can be found in *PYNQ Libraries*.

### Rebuilding the Overlay

The process to rebuild the logictools overlay is similar to the base overlay.

All source code for the hardware blocks is provided. Each block can also be reused standalone in a custom overlay.

The source files for the logictools IP can be found in:

```
<PYNQ Repository>/boards/ip
```

The project files for the logictools overlay can be found here:

```
<PYNQ Repository>/boards/<board_name>/logictools
```

### Linux

To rebuild the overlay, source the Xilinx tools first. Then assuming PYNQ has been cloned:

```
cd <PYNQ Repository>/boards/Pynq-Z1/logictools
make
```

**Windows**

To rebuild from the Vivado GUI, open Vivado. In the Vivado Tcl command line window, change to the correct directory, and source the Tcl files as indicated below.

Assuming PYNQ has been cloned:

```
cd <PYNQ Repository>/boards/Pynq-Z1/logictools
source ./build_logictools_ip.tcl
source ./logictools.tcl
```

To build from the command line, open the Vivado 2017.4 Tcl Shell, and run the following:

```
cd <PYNQ Repository>/boards/Pynq-Z1/logictools
vivado -mode batch -source build_logictools_ip.tcl
vivado -mode batch -source logictools.tcl
```

Note that you must change to the overlay directory, as the .tcl files has relative paths that will break if sourced from a different location.

Other third party overlays may also be available for this board. See the PYNQ community webpage for details of third party overlays and other resources.

### 2.4.4  PYNQ-Z2 Overlays

The PYNQ-Z2 board has the following features:

- Zynq XC7Z020-1CLG400C
- 512MB DDR3
- 1G Ethernet
- USB 2.0
- MicroSD
- Uart
- ADAU1761 Audio Codec with 3.5mm HP/Mic and line-in jacks
- 2x HDMI (can be used as input or output)
- 4 push-buttons
- 2 slide switches
- 4 LEDs
- 2 RGB LEDs
- 2x Pmod ports
- 1x Arduino header
- 1x RaspberryPi header

Note that 8 pins of the RaspberryPi header are shared with one of the Pmod ports.

For details on the PYNQ-Z2 board including reference manual, schematics, constraints file (xdc), see the PYNQ-Z2 webpage

The following overlays are include by default in the PYNQ image for the PYNQ-Z2 board:

**Base Overlay**

The purpose of the base overlay design is to allow PYNQ to use peripherals on a board out-of-the-box. The design includes hardware IP to control peripherals on the target board, and connects these IP blocks to the Zynq PS. If a base overlay is available for a board, peripherals can be used from the Python environment immediately after the system boots.

Board peripherals typically include GPIO devices (LEDs, Switches, Buttons), Video, Audio, and other custom interfaces.

As the base overlay includes IP for the peripherals on a board, it can also be used as a reference design for creating new customized overlays.

In the case of general purpose interfaces, for example Pmod or Arduino headers, the base overlay may include a PYNQ MicroBlaze. A PYNQ MicroBlaze allows control of devices with different interfaces and protocols on the same port without requiring a change to the programmable logic design.

See *PYNQ Libraries* for more information on PYNQ MicroBlazes.

**PYNQ-Z2 Block Diagram**



The base overlay on PYNQ-Z2 includes the following hardware:

- HDMI (Input and Output)
- Audio codec
- User LEDs, Switches, Pushbuttons

- 2x Pmod PYNQ MicroBlaze

- Arduino PYNQ MicroBlaze

- RPi (Raspberry Pi) PYNQ MicroBlaze

- 4x Trace Analyzer (PMODA, PMODB, ARDUINO, RASPBERRYPI)

### HDMI

The PYNQ-Z2 has HDMI in and HDMI out ports. The HDMI interfaces are connected directly to PL pins. i.e. There is no external HDMI circuitry on the board. The HDMI interfaces are controlled by HDMI IP in the programmable logic.

The HDMI IP is connected to PS DRAM. Video can be streamed from the HDMI *in* to memory, and from memory to HDMI *out*. This allows processing of video data from python, or writing an image or Video stream from Python to the HDMI out.

Note that while Jupyter notebook supports embedded video, video captured from the HDMI will be in raw format and would not be suitable for playback in a notebook without appropriate encoding.

### HDMI In

The HDMI in IP can capture standard HDMI resolutions. After a HDMI source has been connected, and the HDMI controller for the IP is started, it will automatically detect the incoming data. The resolution can be read from the HDMI Python class, and the image data can be streamed to the PS DRAM.

### HDMI Out

The HDMI out IP supports the following resolutions:

- 640x480

- 800x600

- 1280x720 (720p)

- 1280x1024

- 1920x1080 (1080p)*

*While the Pynq-Z2 cannot meet the official HDMI specification for 1080p, some HDMI devices at this resolution may work.

Data can be streamed from the PS DRAM to the HDMI output. The HDMI Out controller contains framebuffers to allow for smooth display of video data.

See example video notebooks in the `<Jupyter Dashboard>/base/video` directory on the board.

### Audio

The PYNQ-Z2 base overlay supports line in, and Headphones out/Mic. The audio source can be selected, either line-in or Mic, and the audio in to the board can be either recorded to file, or played out on the headphone output.

### User IO

The PYNQ-Z2 board includes two tri-color LEDs, 2 switches, 4 push buttons, and 4 individual LEDs. These IO are connected directly to Zynq PL pins. In the PYNQ-Z2 base overlay, these IO are routed to the PS GPIO, and can be controlled directly from Python.

### PYNQ MicroBlaze

PYNQ MicroBlazes are dedicated MicroBlaze soft-processor subsystems that allow peripherals with different IO standards to be connected to the system on demand. This allows a software programmer to use a wide range of peripherals with different interfaces and protocols. By using a PYNQ MicroBlaze, the same overlay can be used to support different peripheral without requiring a different overlay for each peripheral.

The PYNQ-Z2 has three types of PYNQ MicroBlaze: *Pmod*, *Arduino*, and *RPi* (Raspberry Pi), connecting to each type of corresponding interface. There is one instance of the Arduino, and one instance of the RPi PYNQ MicroBlaze, and two instances of the Pmod PYNQ MicroBlaze in the base overlay.

Each physical interface has a different number of pins and can support different sets of peripherals. Each PYNQ MicroBlaze has the same core architecture, but can have different IP configurations to support the different sets of peripheral and interface pins.

Note that because the 8 data pins of PmodA are shared with the lower 8 data pins of the RPi header, the `base.select_pmoda()` function must be called before loading an application on PmodA, and `base.select_pmoda()` must be called before loading an application on the RPi PYNQ MicroBlaze.

PYNQ MicroBlaze block diagram and examples can be found in *MicroBlaze Subsystem*.

### Trace Analyzer

Trace analyzer blocks are connected to the interface pins for the two Pmod PYNQ MicroBlazes, the Arduino and RPi PYNQ MicroBlazes. The trace analyzer can capture IO signals and stream the data to the PS DRAM for analysis in the Python environment.

Using the Python Wavedrom package, the signals from the trace analyzer can be displayed as waveforms in a Jupyter notebook.

On the base overlay, the trace analyzers are controlled by PS directly. In fact, on other overlays, the trace analyzers can also be controlled by PYNQ MicroBlaze.

See the example notebook in the `<Jupyter Dashboard>/base/trace` directory on the board.

### Python API

The Python API for the peripherals in the base overlay is covered in *PYNQ Libraries*. Example notebooks are also provided on the board to show how to use the base overlay.

### Rebuilding the Overlay

The project files for the overlays can be found here:

```
<PYNQ repository>/boards/<board>/base
```

### Linux

A Makefile is provided to rebuild the base overlay in Linux. The Makefile calls two tcl files. The first Tcl files compiles any HLS IP used in the design. The second Tcl builds the overlay.

To rebuild the overlay, source the Xilinx tools first. Then assuming PYNQ has been cloned:

```
cd <PYNQ repository>/boards/Pynq-Z2/base
make
```

### Windows

In Windows, the two Tcl files can be sourced in Vivado to rebuild the overlay. The Tcl files to rebuild the overlay can be sourced from the Vivado GUI, or from the Vivado Tcl Shell (command line).

To rebuild from the Vivado GUI, open Vivado. In the Vivado Tcl command line window change to the correct directory, and source the Tcl files as indicated below.

Assuming PYNQ has been cloned:

```
cd <PYNQ repository>/boards/Pynq-Z2/base
source ./build_base_ip.tcl
source ./base.tcl
```

To build from the command line, open the Vivado 2017.4 Tcl Shell, and run the following:
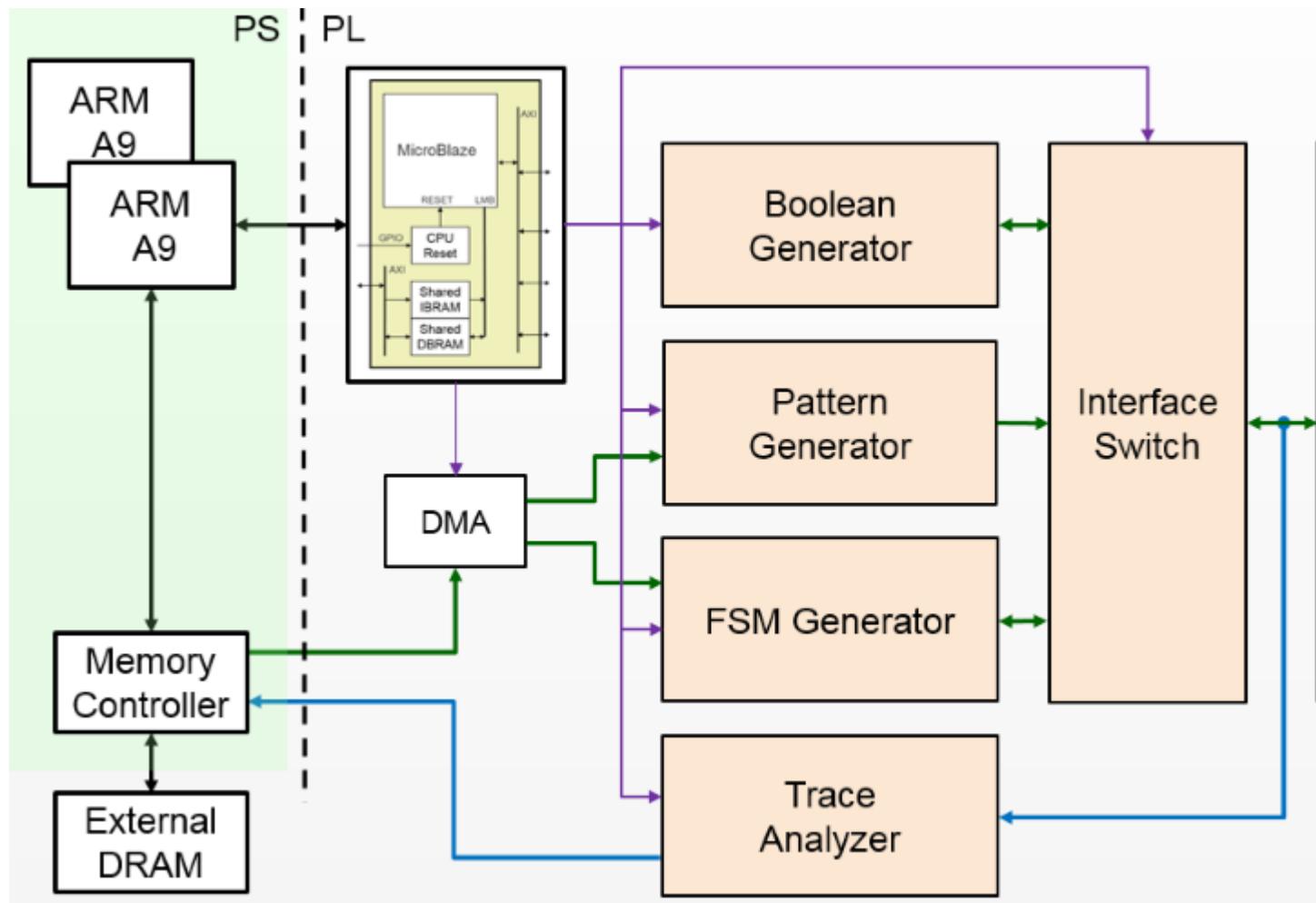
```
cd <PYNQ repository>/boards/Pynq-Z2/base
vivado -mode batch -source build_base_ip.tcl
vivado -mode batch -source base.tcl
```

Note that you must change to the overlay directory, as the tcl files has relative paths that will break if sourced from a different location.

### Logictools Overlay

The logictools overlay consists of programmable hardware blocks to connect to external digital logic circuits. Finite state machines, Boolean logic functions and digital patterns can be generated from Python. A programmable switch connects the inputs and outputs from the hardware blocks to external IO pins. The logictools overlay can also has a trace analyzer to capture data from the IO interface for analysis and debug.

**Logictools base diagram**



The logictools IP includes four main hardware blocks:

- Pattern Generator
- FSM Generator
- Boolean Generator
- Trace Analyzer
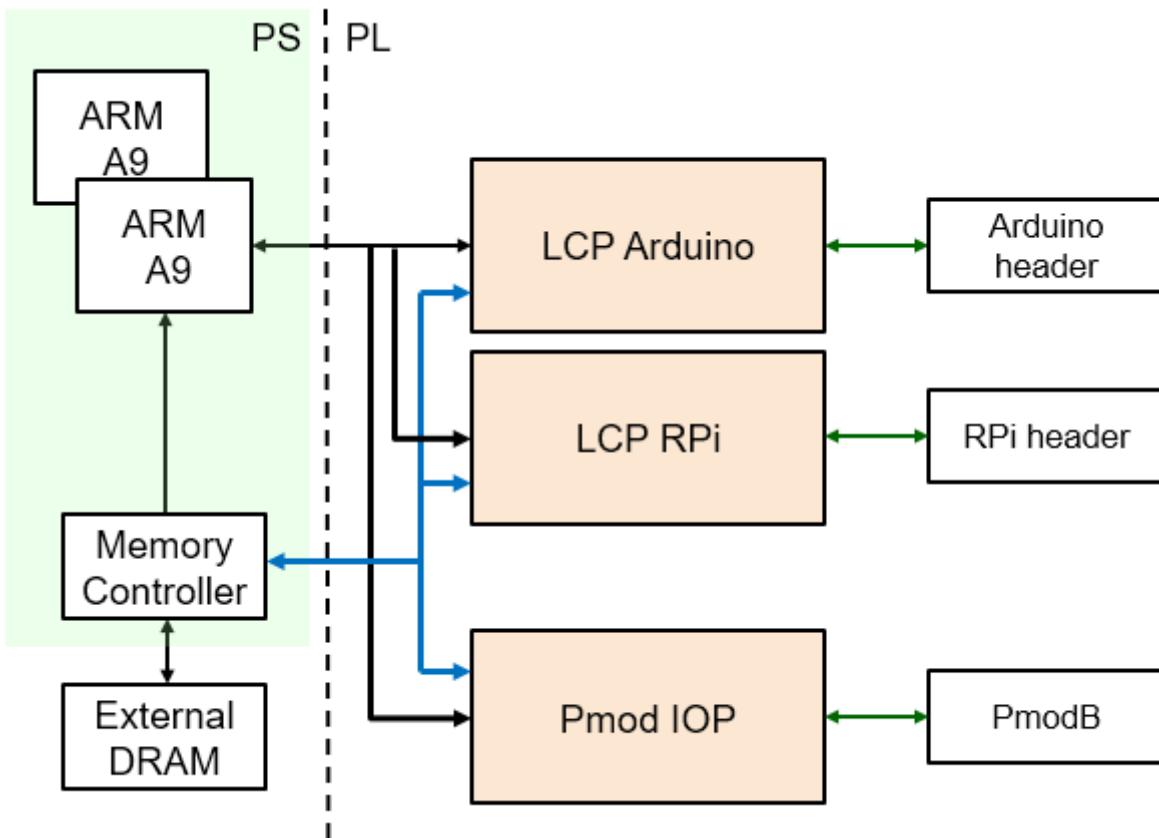
Each block is configured using a textual description specified in Python. No compilation of the configuration is required. This means a configuration can be loaded directly to the generator and run immediately.

**PYNQ-Z2 logic tools**

The PYNQ-Z2 logictools overlay has two instances of the logictools LCP (Logic Control Processor); one connected to the Arduino header, and the other connected to the RPi (Raspberry Pi) header.

The Arduino header has 20 pins, and the RPi has 26 pins that can be used as GPIO to the LCP.

The 4 LEDs, and 4 pushbuttons can also be connected to either LCP, extending the number of inputs available. Note that the LEDs and pushbuttons are shared, and can only be used by one LCP at a time.

The overlay also includes a Pmod IOP connected to PmodB. This is the same Pmod IOP that is used in the base overlay.

### Pattern Generator

The *Pattern Generator* can be configured to generate and stream arbitrary digital patterns to the external IO pins. The Pattern Generator can be used as a stimulus to test or control an external circuit.

### Finite State Machine (FSM) Generator

The *FSM Generator* can create a finite state machine from a Python description. The inputs and outputs and states of the FSM can be connected to external IO pins.

### Boolean Generator

The *Boolean Generator* can create independent combinatorial Boolean logic functions. The external IO pins are used as inputs and outputs to the Boolean functions.

### Trace Analyzer

The *Trace Analyzer* can capture IO signals and stream the data to the PS DRAM for analysis in the Python environment. The Trace Analyzer can be used standalone to capture external IO signals, or used in combination with the other three logictools functions to monitor data to and from the other blocks. E.g. the trace analyzer can be used with the pattern

generator to verify the data sent to the external pins, or with the FSM to check the input, output or states to verify or debug a design.

**PYNQ MicroBlaze**

A PYNQ MicroBlaze is used to control all the generators and analyzers. The PYNQ MicroBlaze subsystem on logictools overlay also manages contiguous memory buffers, configures the clock frequency, and keeps track of the generator status. For more information, please see *PYNQ Libraries*.

**Python API**

The API for the logictools generators and trace analyzer can be found in *PYNQ Libraries*.

**Rebuilding the Overlay**

The process to rebuild the logictools overlay is similar to the base overlay.

All source code for the hardware blocks is provided. Each block can also be reused standalone in a custom overlay.

The source files for the logictools IP can be found in:

```
<PYNQ Repository>/boards/ip
```

The project files for the logictools overlay can be found here:

```
<PYNQ Repository>/boards/<board_name>/logictools
```

**Linux**

To rebuild the overlay, source the Xilinx tools first. Then assuming PYNQ has been cloned:

```
cd <PYNQ Repository>/boards/Pynq-Z2/logictools
make
```

**Windows**

To rebuild from the Vivado GUI, open Vivado. In the Vivado Tcl command line window, change to the correct directory, and source the Tcl files as indicated below.

Assuming PYNQ has been cloned:

```
cd <PYNQ Repository>/boards/Pynq-Z2/logictools
source ./build_logictools_ip.tcl
source ./logictools.tcl
```

To build from the command line, open the Vivado 2017.4 Tcl Shell, and run the following:

```
cd <PYNQ Repository>/boards/Pynq-Z2/logictools
vivado -mode batch -source build_logictools_ip.tcl
vivado -mode batch -source logictools.tcl
```

Note that you must change to the overlay directory, as the .tcl files has relative paths that will break if sourced from a different location.

Other third party overlays may also be available for this board. See the PYNQ community webpage for details of third party overlays and other resources.

## 2.4.5 ZCU104 Overlays

The ZCU104 board has the following features:

Device

- Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC

Configuration

- USB-JTAG FT4232H
- Dual Quad-SPI flash memory
- MicroSD Card

Memory

- PS DDR4 64-bit Component
- Quad-SPI flash
- Micro SD card slot

Control & I/O

- 4x directional pushbuttons
- DIP switches
- PMBUS, clocks, and I2C bus switching
- USB2/3

Expansion Connectors

- FMC LPC (1x GTH)
- 3 PMOD connectors
- PL DDR4 SODIMM Connector – 64 bit

Communication & Networking

- USB-UARTs with FT4232H JTAG/3xUART Bridge
- RJ-45 Ethernet connector
- SATA (M.2) for SSD access

Display

- HDMI 2.0 video input and output (3x GTH)
- DisplayPort (2x GTR)

Power

- 12V wall adaptor or ATX

For details on the ZCU104 board including reference manual, schematics, constraints file (xdc), see the Xilinx ZCU104 webpage

The following overlays are include by default in the PYNQ image for the ZCU104 board:

### Base Overlay

The purpose of the *base* overlay design for any PYNQ supported board is to allow peripherals on a board to be used out-of-the-box.

The design includes hardware IP to control peripherals on the target board, and connects these IP blocks to the Zynq PS. If a base overlay is available for a board, peripherals can be used from the Python environment immediately after the system boots.
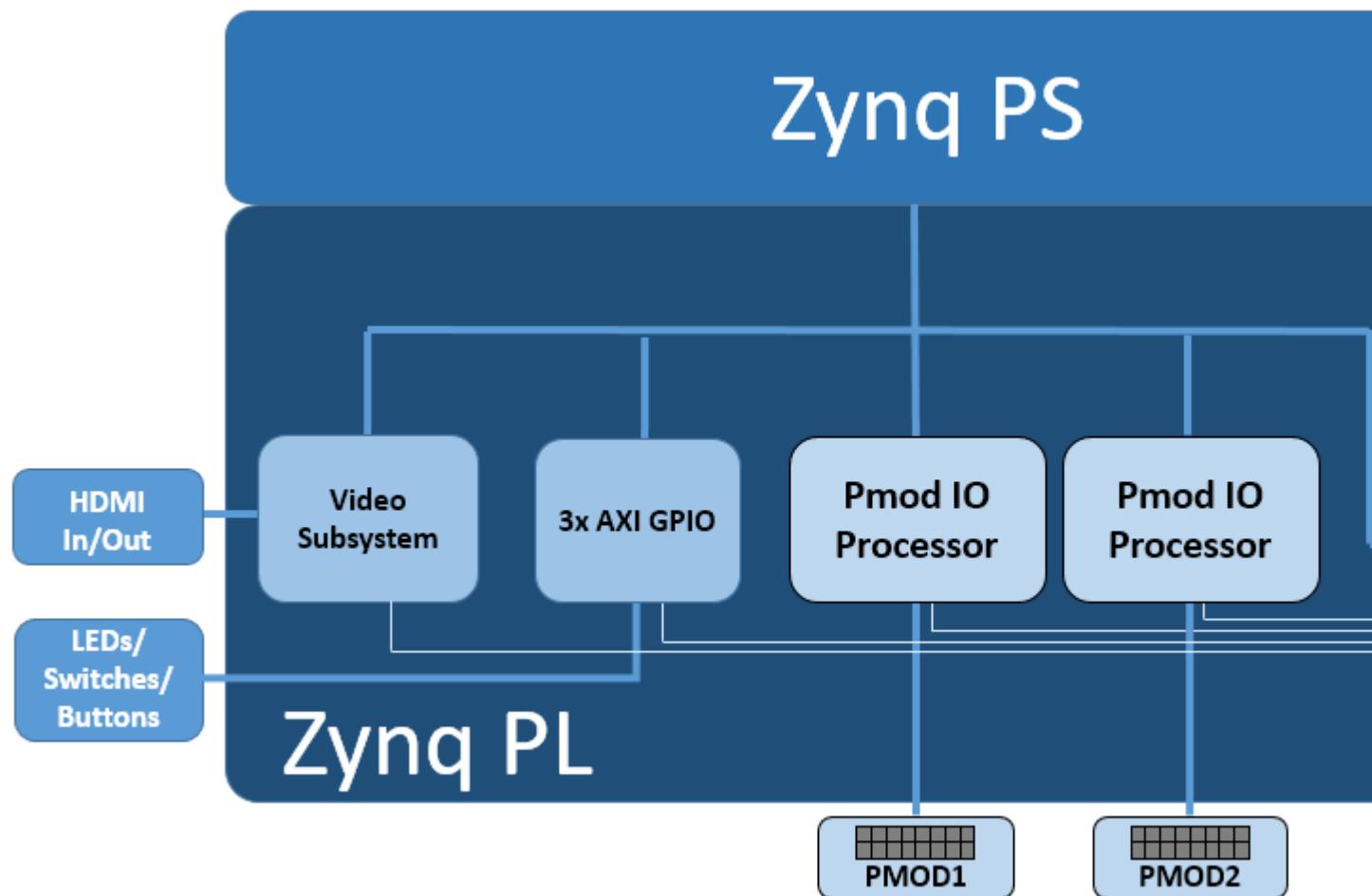
Board peripherals typically include GPIO devices (LEDs, Switches, Buttons), Video, and other custom interfaces.

As the base overlay includes IP for the peripherals on a board, it can also be used as a reference design for creating new customized overlays.

In the case of general purpose interfaces, for example Pmod or Arduino headers, the base overlay may include a PYNQ MicroBlaze. A PYNQ MicroBlaze allows control of devices with different interfaces and protocols on the same port without requiring a change to the programmable logic design.

See *PYNQ Libraries* for more information on PYNQ MicroBlazes.

**ZCU104 Block Diagram**



The base overlay on ZCU104 includes the following hardware:

- DisplayPort & HDMI (Input and Output)
- User LEDs, Switches, Pushbuttons
- 2x Pmod PYNQ MicroBlaze

**HDMI**

The ZCU104 has 2x HDMI ports supporting HDMI 2.0 video input and output. The HDMI interfaces are controlled by HDMI IP in the programmable logic.

The HDMI IP is connected through a video DMA to PS DRAM. Video can be streamed from the HDMI *in* to memory, and from memory to HDMI *out*. This allows processing of video data from python, or writing an image or Video stream from Python to the HDMI out.

Note that the ZCU104 also has a DisplayPort connected to the PS. While the Display port is not part of the Overlay (as it is always connected) video data can be streamed from the HDMI PL sources to the DisplayPort.

Note that while Jupyter notebook supports embedded video, video captured from the HDMI will be in raw format and would not be suitable for playback in a notebook without appropriate encoding.

### HDMI In

The HDMI in IP can capture standard HDMI resolutions. After a HDMI source has been connected, and the HDMI controller for the IP is started, it will automatically detect the incoming data. The resolution can be read from the HDMI Python class, and the image data can be streamed to the PS DRAM.

### HDMI Out

Data can be streamed from the PS DRAM to the HDMI output. The HDMI Out controller contains framebuffers to allow for smooth display of video data.

See example video notebooks in the `<Jupyter Dashboard>/base/video` directory on the board.

### User IO

The board has 4x user LEDs, 4x dip-switches, 4x push buttons. These IO are controlled via AXI GPIO controllers in the PL. In the ZCU104 base overlay, these IO are routed to the PS GPIO, and can be controlled directly from Python.

### PYNQ MicroBlaze

PYNQ MicroBlazes are dedicated MicroBlaze soft-processor subsystems that allow peripherals with different IO standards to be connected to the system on demand. This allows a software programmer to use a wide range of peripherals with different interfaces and protocols. By using a PYNQ MicroBlaze, the same overlay can be used to support different peripheral without requiring a different overlay for each peripheral.

The ZCU has 2x Pmod PYNQ MicroBlaze, one connected to PMOD 0, and the other to PMOD 1. connecting to each type of corresponding interface. Note that PMOD 2 is connected to the PS I2C and is not available as a general purpose Pmod port.

PYNQ MicroBlaze block diagram and examples can be found in *MicroBlaze Subsystem*.

### Python API

The Python API for the peripherals in the base overlay is covered in *PYNQ Libraries*. Example notebooks are also provided on the board to show how to use the base overlay.

### Rebuilding the Overlay

The project files for the overlays can be found here:

```
<PYNQ repository>/boards/<board>/base
```

### Linux

A Makefile is provided to rebuild the base overlay in Linux. The Makefile calls two tcl files. The first Tcl files compiles any HLS IP used in the design. The second Tcl builds the overlay.

To rebuild the overlay, source the Xilinx tools first. Then assuming PYNQ has been cloned:

```
cd <PYNQ repository>/boards/ZCU104/base
make
```

### Windows

In Windows, the two Tcl files can be sourced in Vivado to rebuild the overlay. The Tcl files to rebuild the overlay can be sourced from the Vivado GUI, or from the Vivado Tcl Shell (command line).

To rebuild from the Vivado GUI, open Vivado. In the Vivado Tcl command line window change to the correct directory, and source the Tcl files as indicated below.

Assuming PYNQ has been cloned:

```
cd <PYNQ repository>/boards/ZCU104/base
source ./build_base_ip.tcl
source ./base.tcl
```

To build from the command line, open the Vivado Tcl Shell, and run the following:

```
cd <PYNQ repository>/boards/ZCU104/base
vivado -mode batch -source build_base_ip.tcl
vivado -mode batch -source base.tcl
```

Note that you must change to the overlay directory, as the tcl files has relative paths that will break if sourced from a different location.

Other third party overlays may also be available for this board. See the PYNQ community webpage for details of third party overlays and other resources.

## 2.5 PYNQ Libraries

Typical embedded systems support a fixed combination of peripherals (e.g. SPI, IIC, UART, Video, USB ). There may also be some GPIO (General Purpose Input/Output pins) available. The number of GPIO available in a CPU based embedded system is typically limited, and the GPIO are also controlled by the main CPU. As the main CPU which is managing the rest of the system, GPIO performance is usually limited.

Zynq platforms usually have many more IO pins available than a typical embedded system. Dedicated hardware controllers and additional soft processors can be implemented in the PL and connected to external interfaces. This means performance on these interfaces can be much higher than other embedded system.

PYNQ runs on Linux which uses the following Zynq PS peripherals by default: SD Card to boot the system and host the Linux file system, Ethernet to connect to Jupyter notebook, UART for Linux terminal access, and USB.

The USB port and other standard interfaces can be used to connect off-the-shelf USB and other peripherals to the Zynq PS where they can be controlled from Python/Linux. The PYNQ image currently includes drivers for the most commonly used USB webcams, WiFi peripherals, and other standard USB devices.

Other peripherals can be connected to and accessed from the Zynq PL. E.g. HDMI, Audio, Buttons, Switches, LEDs, and general purpose interfaces including Pmods, and Arduino. As the PL is programmable, an overlay which provides controllers for these peripherals or interfaces must be loaded before they can be used.

A library of hardware IP is included in Vivado which can be used to connect to a wide range of interface standards and protocols. PYNQ provides a Python API for a number of common peripherals including Video (HDMI in and Out), GPIO devices (Buttons, Switches, LEDs), and sensors and actuators. The PYNQ API can also be extended to support additional IP.

Zynq platforms usually have one or more *headers* or *interfaces* that allow connection of external peripherals, or to connect directly to the Zynq PL pins. A range of off-the-shelf peripherals can be connected to Pmod and Arduino interfaces. Other peripherals can be connected to these ports via adapters, or with a breadboard. Note that while a peripheral can be physically connected to the Zynq PL pins, a controller must be built into the overlay, and a software driver provided, before the peripheral can be used.

The PYNQ libraries provide support for the PynqMicroBlaze subsystem, allowing pre-compiled applications to be loaded, and new applications to be creating and compiled from Jupyter.

PYNQ also provides support for low level control of an overlay including memory-mapped IO read/write, memory allocation (for example, for use by a PL master), control and management of an overlay (downloading an overlay, reading IP in an overlay), and low level control of the PL (downloading a bitstream).

### 2.5.1 IP

#### Audio

The Audio module provides methods to read audio from the input microphone, play audio to the output speaker, or read and write audio files. The audio module connects to the audio IP subsystem in in overlay to capture and playback data. The audio module is intended to support different IP subsystems. It currently supports the line-in, HP/Mic with the ADAU1761 codec on the PYNQ-Z2 and the Pulse Width Modulation (PWM) mono output and Pulse Density Modulated (PDM) microphone input on the PYNQ-Z1 board.

#### Examples

Both the *Base Overlay* and the *Base Overlay* contain a single Audio instance: *audio*. After the overlay is loaded this instance can be accessed as follows:

#### PYNQ-Z1

```
from pynq.overlays.base import BaseOverlay
base = BaseOverlay("base.bit")
pAudio = base.audio

pAudio.load("/home/xilinx/pynq/lib/tests/pynq_welcome.pdm")
pAudio.play()
```

#### PYNQ-Z2

```
from pynq.overlays.base import BaseOverlay
base = BaseOverlay("base.bit")
pAudio = base.audio
pAudio.load("/home/xilinx/jupyter_notebooks/base/audio/data/recording_0.wav")

pAudio.play()
```

(Note the PYNQ-Z1 supports direct playback of PDM out, and the PYNQ-Z2 supports Wav.)

More information about the Audio module and the API for reading and writing audio interfaces, or loading and saving audio files can be found in the *pynq.lib.audio Module* section.

For more examples see the Audio notebook on your PYNQ-Z1 or PYNQ-Z2 board: at:

```
<Jupyter Home>/base/audio/audio_playback.ipynb
```

### AxiGPIO

The AxiGPIO class provides methods to read, write, and receive interrupts from external general purpose peripherals such as LEDs, buttons, switches connected to the PL using AXI GPIO controller IP.

### Block Diagram

The AxiGPIO module controls instances of the AXI GPIO controller in the PL. Each AXI GPIO can have up to two channels each with up to 32 pins.



The `read()` and `write()` methods are used to read and write data on a channel (all of the GPIO).
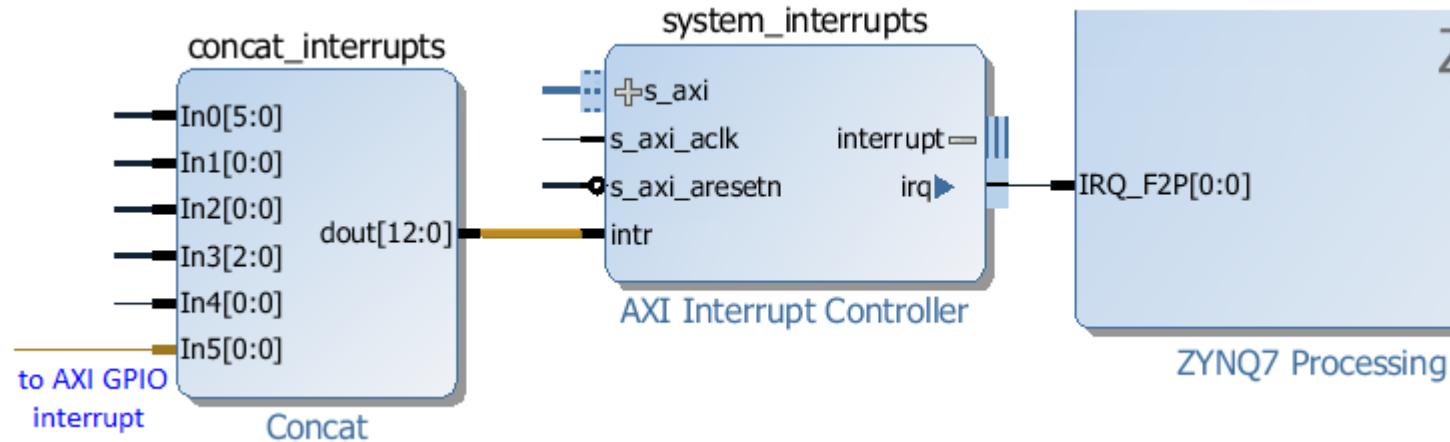
`setdirection()` and `setlength()` can be used to configure the IP. The direction can be 'in', 'out', and 'inout'.

By default the direction is 'inout'. Specifying 'in' or 'out' will only allow read and writes to the IP respectively, and trying to *read* an 'out' or *write* an 'in' will cause an error.

The length can be set to only write a smaller range of the GPIO.

The GPIO can also be treated like an array. This allows specific bits to be set, and avoids the need to use a bit mask.

The interrupt signal, *ip2intc_irpt* from the AXI GPIO can be connected directly to an AXI interrupt controller to cause interrupts in the PS. More information about AsyncIO and Interrupts can be found in the *PYNQ and Asyncio* section.

The LED, Switch, Button and RGBLED classes extend the AxiGPIO controller and are customized for the corresponding peripherals. These classes expect an AXI GPIO instance called `[led|switch|button|rgbleds]_gpio` to exist in the overlay used with this class.

### Examples

This example is for illustration, to show how to use the AxiGPIO class. In practice, the LED, Button, Switches, and RGBLED classes may be available to extend the AxiGPIO class should be used for these peripherals in an overlay.

After an overlay has been loaded, an AxiGPIO instance can be instantiated by passing the name of the AXI GPIO controller to the class.

```python
from pynq import Overlay
from pynq.lib import AxiGPIO
ol = Overlay("base.bit")

led_ip = ol.ip_dict['gpio_leds']
switches_ip = ol.ip_dict['gpio_switches']
leds = AxiGPIO(led_ip).channel1
switches = AxiGPIO(switches_ip).channel1
```

Simple read and writes:

```python
mask = 0xffffffff
leds.write(0xf, mask)
switches.read()
```

Using AXI GPIO as an array:

```python
switches.setdirection("in")
switches.setlength(3)
switches.read()
```

More information about the AxiGPIO module and the API for reading, writing and waiting for interrupts can be found in the *pynq.lib.axigpio Module* sections.

For more examples see the "Buttons and LEDs demonstration" notebook for the PYNQ-Z1/PYNQ-Z2 board at:

```
<Jupyter Home>/base/board/board_btns_leds.ipynb
```

The same notebook may be found in the corresponding folder in the GitHub repository.

### AxiIIC

The AxiIIC class provides methods to read from , and write to an AXI IIC controller IP.

The `send()` and `receive()` methods are used to read and write data.

```
send(address, data, length, option=0)
```

- address is the address of the IIC peripheral

- data is an array of bytes to be sent to the IP

- length is the number of bytes to be transferred

- option allows an IIC *repeated start*

```
receive(address, data, length, option=0)
```

- address is the address of the IIC peripheral

- data is an array of bytes to receive data from the IP

- length is the number of bytes to be received

- option allows an IIC *repeated start*

More information about the AxiIIC module and the API for reading, writing and waiting can be found in the *pynq.lib.iic Module* sections.
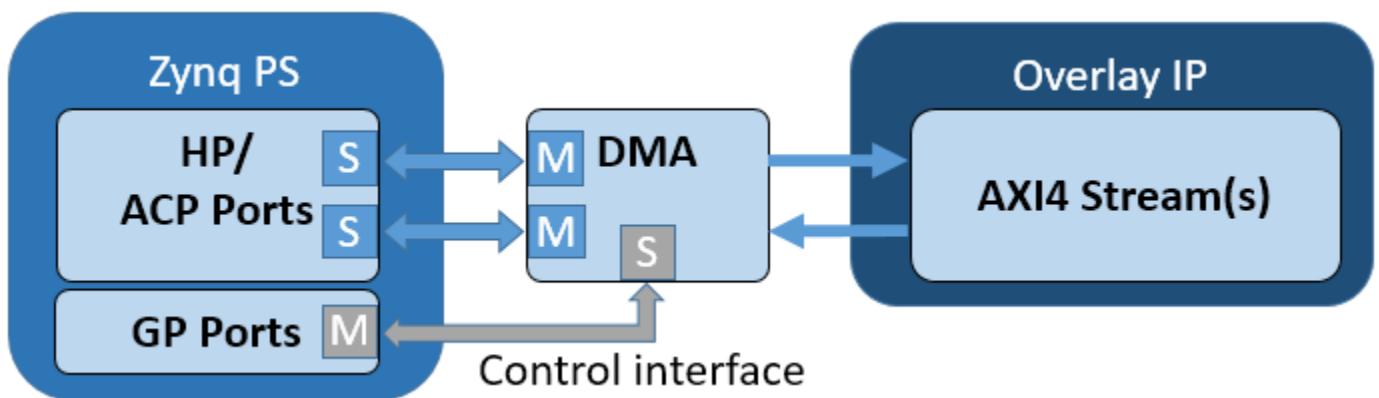
## DMA

PYNQ supports the AXI central DMA IP with the PYNQ *DMA* class. DMA can be used for high performance burst transfers between PS DRAM and the PL.

The *DMA* class supports simple mode only.

## Block Diagram

The DMA has an AXI lite control interface, and a read and write channel which consist of a AXI master port to access the memory location, and a stream port to connect to an IP.



The read channel will read from PS DRAM, and write to a stream. The write channel will read from a stream, and write back to PS DRAM.

Note that the DMA expects any streaming IP connected to the DMA (write channel) to set the AXI TLAST signal when the transaction is complete. If this is not set, the DMA will never complete the transaction. This is important when using HLS to generate the IP - the TLAST signal must be set in the C code.

## Examples

This example assumes the overlay contains an AXI Direct Memory Access IP, with a read channel (from DRAM), and an AXI Master stream interface (for an output stream), and the other with a write channel (to DRAM), and an AXI Slave stream interface (for an input stream).

In the Python code, two contiguous memory buffers are created using `allocate`. The DMA will read the input_buffer and send the data to the AXI stream master. The DMA will write back to the output_buffer from the AXI stream slave.

---

The AXI streams are connected in loopback so that after sending and receiving data via the DMA the contents of the input buffer will have been transferred to the output buffer.

Note that when instantiating a DMA, the default maximum transaction size is 14-bits (i.e. 2^14 = 16KB). For larger DMA transactions, make sure to increase this value when configuring the DMA in your Vivado IPI design.

In the following example, let's assume the *example.bit* contains a DMA IP block with both send and receive channels enabled.

```python
import numpy as np
from pynq import allocate
from pynq import Overlay

overlay = Overlay('example.bit')
dma = overlay.axi_dma

input_buffer = allocate(shape=(5,), dtype=np.uint32)
output_buffer = allocate(shape=(5,), dtype=np.uint32)
```

Write some data to the array:

```python
for i in range(5):
    input_buffer[i] = i
```

```
Input buffer will contain:  [0 1 2 3 4]
```

Transfer the input_buffer to the send DMA, and read back from the receive DMA to the output buffer. The wait() method ensures the DMA transactions have completed.

```python
dma.sendchannel.transfer(input_buffer)
dma.recvchannel.transfer(output_buffer)
dma.sendchannel.wait()
dma.recvchannel.wait()
```
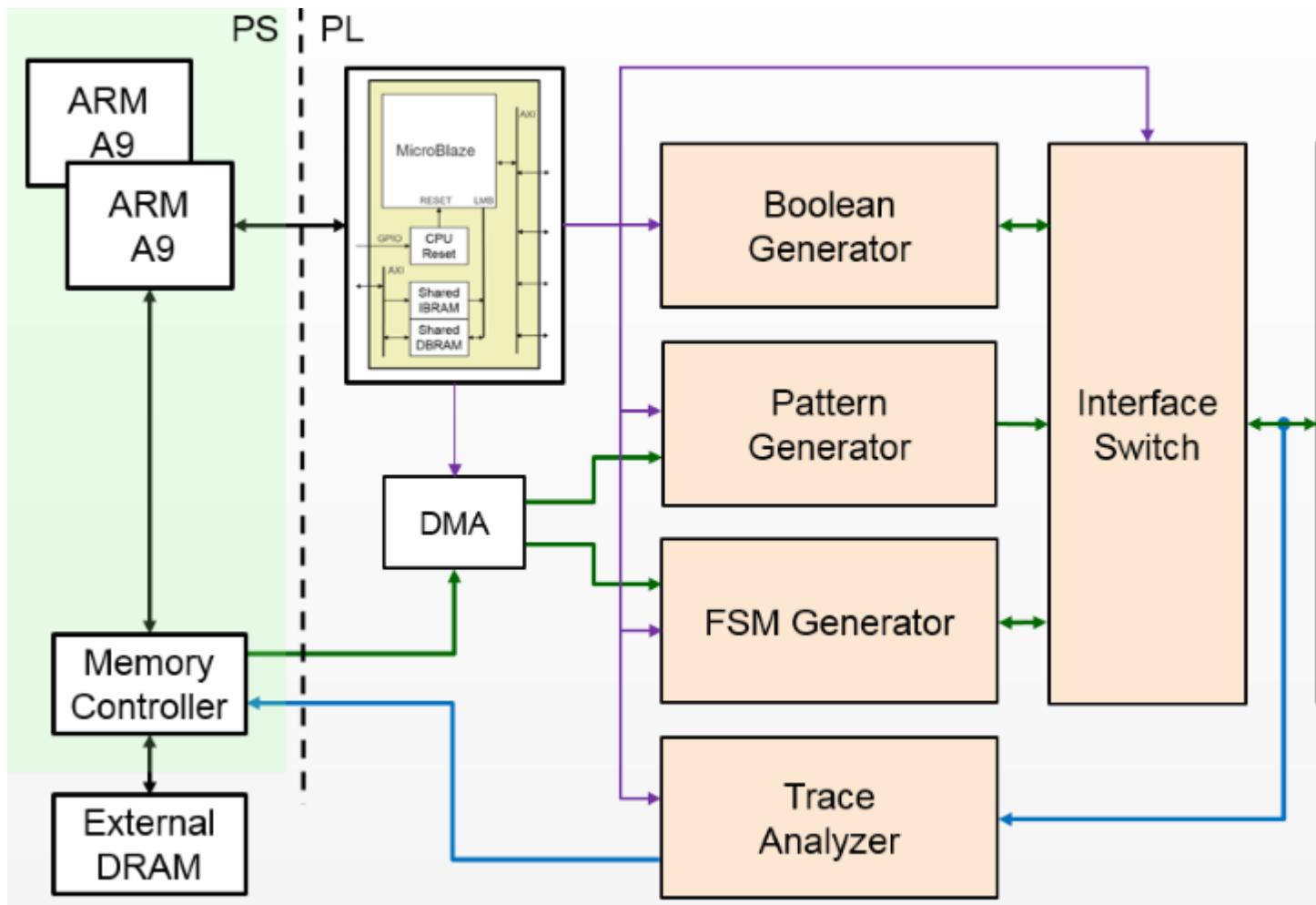
```
Output buffer will contain: [0 1 2 3 4]
```

More information about the DMA module can be found in the *pynq.lib.dma Module* sections

### Logictools

The logictools subpackage contains drivers for the Trace Analyzer, and the three PYNQ hardware generators: Boolean Generator, FSM Generator, and Pattern Generator.

**Block Diagram**
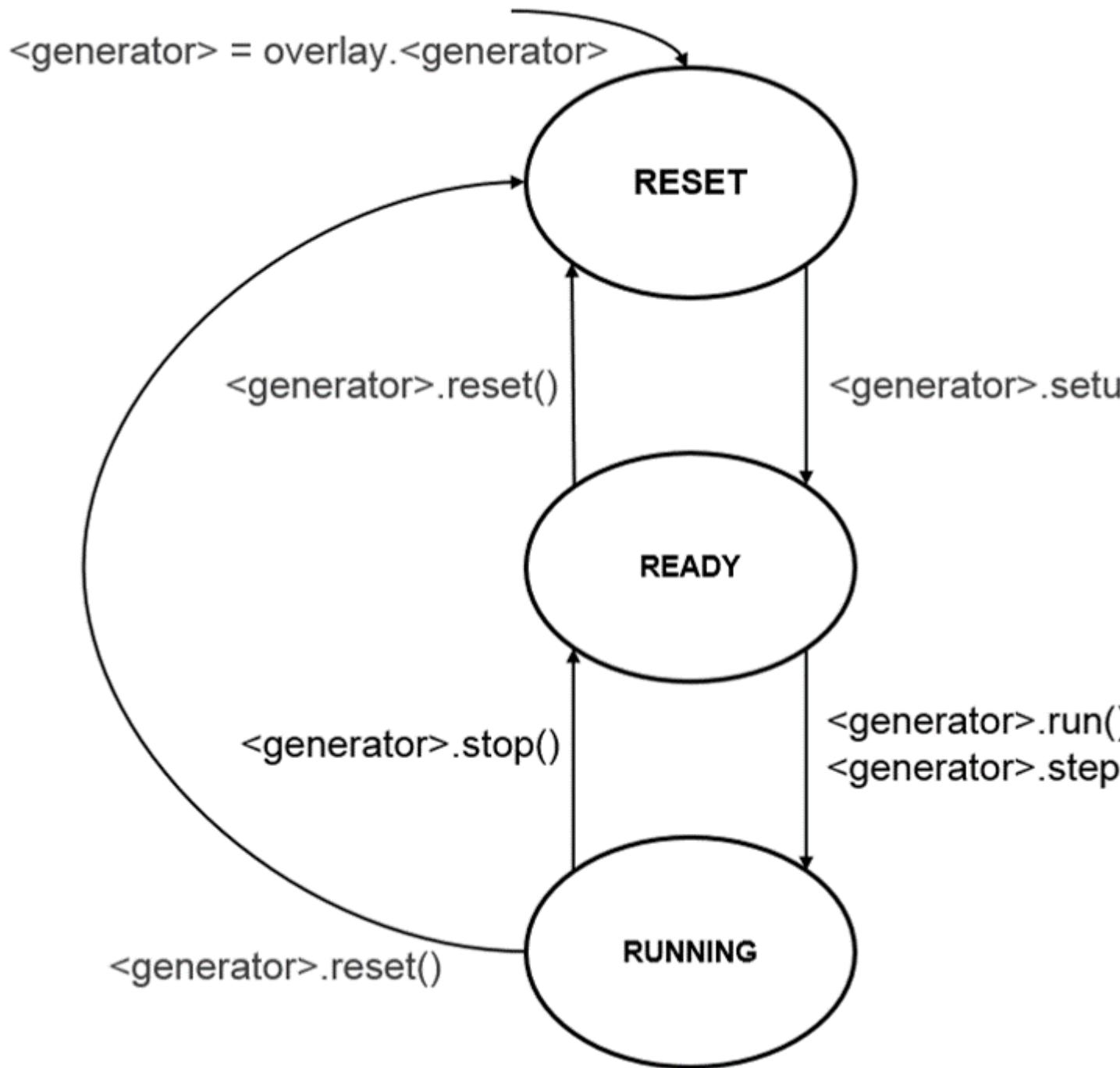


**States**

The basic operation of the main hardware blocks in the logictools overlay is the same. A set of methods which is common to all blocks is used to control basic operations, `setup()`, `run()`, `step()`, `stop()`, `reset()`. The operation of these methods will be described later.

Each block may have additional unique methods to provide functionality specific to that block.

The state diagram for the blocks is shown below:

Any one of these hardware blocks, or any combination can be configured and run synchronously following the state diagram above.

### RESET

This is the state a block will start in after the overlay is loaded. A block will remain in the reset state until it has been configured using the `setup()` method. It will return to this state if `reset()` is called.

In the reset state, all IO accessible to the logictools overlay are disconnected from the main logictools hardware

blocks. This prevents the inadvertent driving of any external circuitry that is connected to the board. This is done by configuring the interface switch to disconnect all IO from the internal hardware.

The Pattern Generator contains BRAM to store the pattern to be generated. The BRAM is configured with zeros in this state.

Similarly, the FSM Generator configuration is stored in a BRAM which is also configured with zeros in this state.

### READY

In this state, the generators / analyzer have been configured. The input and output pins that will be connected have been specified, and reserved, but the interface switch has not bee configured to connect these pins to the internal hardware.

### RUNNING

Once the generators are in the ready state, calling run() or step() will move them to the READY state. When moving to this state, the interface switch is configured to connect external IO. The hardware block(s) will start operating in this state.

Running will start the block running in single-shot mode by default. In this mode, the generator will run until enough number of samples are captured by the trace analyzer, or the pattern has completed; then the generator and analyzer both go back to the READY state.

Boolean Generator always runs in continuous mode as a special case.

In continuous mode, the Pattern Generator generates its pattern continuously, looping back to the start when it reaches the end of the pattern. The FSM Generator will continue to run until it is stopped.

### Methods

Each generator / analyzer has the following methods:

- `setup()` - configure the block and prepare Interface Switch configuration
- `run()` - connect IO and start the block running
- `stop()` - disconnect IO and stop the block running
- `step()` - run a single step for the pattern or FSM generator
- `reset()` - clear the block configuration
- `trace()` - enable/disable trace

### setup()

Each block must be configured using the `setup()` method before it can be used. This defines a configuration for the block, and the configuration for the Interface Switch to connect the external IO. Note that the configuration is defined, but the IO are not connected during setup.

### run()

The `run()` method will move a block to the *RUNNING* state and the block will start operating. The specified number of samples will be captured by the Trace Analyzer.

### step()

The `step()` method is similar to `run()`, but instead of running, all the generators are single stepped (advanced one clock cycle) each time the *step* method is called.

When stepping the Pattern Generator, it will step until the end of the configured pattern. It will not loop back to the beginning.

The FSM Generator can be single stepped until a enough samples are captured by the Trace Analyzer.

### stop()

If a block is running, it must be stopped before re-running.

Once a block is stopped, its outputs are disconnected from the external IO, and will only be reconnected when the block is set to run again.

### trace()

Trace is enabled by default for each block. When trace is enabled, the Trace Analyzer will capture trace data for all connected blocks. The `trace()` method can be used to enable/disable the Trace Analyzer for each block.
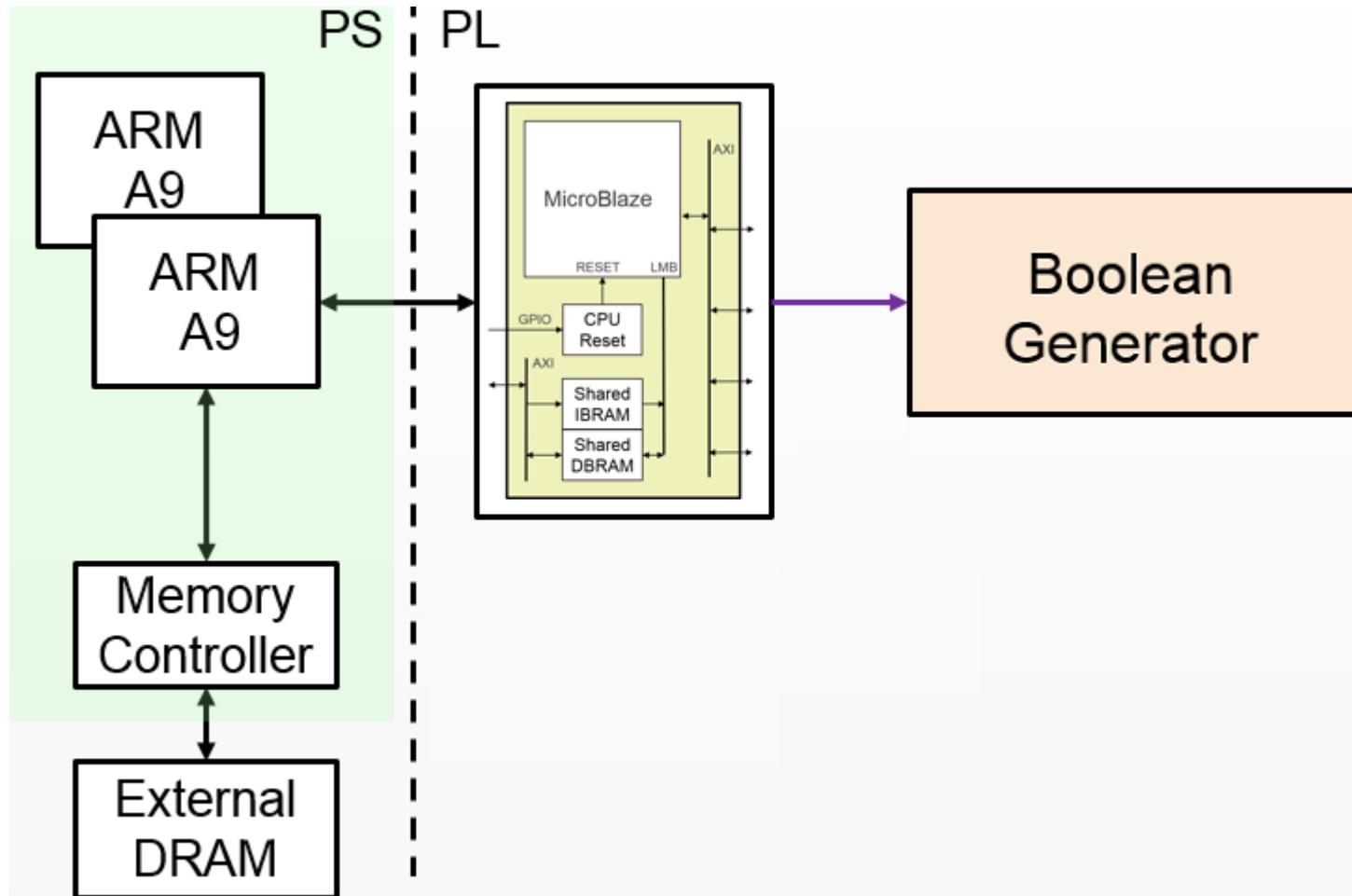
### reset()

This method resets the generator to its initial state. This method needs to be called before changing the configuration for a hardware block.

### Boolean Generator

The Boolean Generator supports up to Boolean functions of up to five inputs on each output pin. AND, OR, NOT, and XOR operators are supported.

**Block Diagram**



On the PYNQ-Z1 the 20 digital pins of the Arduino shield interface (D0 - D19) can be used as inputs or outputs. The 4 pushbuttons (PB0 - PB3) can be used as additional inputs, and the 4 user LEDs (LD0 - LD3) can be used as additional outputs. This gives a maximum of 24 inputs and outputs available to the Boolean Generator, supporting up to 24 Boolean functions.

Boolean functions are specified, as strings.

For example the following specifies that the values of pushbuttons 1 and 0 are XORed to produce the value on LED0:

```
'LD0 = PB0 ^ PB1'
```

Combinatorial Boolean expressions can be defined in a Python list using the expressions & (AND), | (OR), ! (NOT), ^ (XOR).

The expression defines if a pin is used as an input or output.

**Examples**

The following list defines four combinatorial functions on pins D8-11, which are built using combinatorial functions made up of inputs from pins D0-D3. Any pin assigned a value is an output, and any pin used as a parameter in the expression is an input. If a pin is defined as an output, it cannot be used as an input.

```python
from pynq.overlays.logictools import LogicToolsOverlay

logictools = LogicToolsOverlay('logictools.bit')
boolean_generator = logictools.boolean_generator

function_specs = ['D3 = D0 ^ D1 ^ D2']
function_specs.append('D6 = D4 & D5')
```

The function configurations can also be labelled:

```python
function_specs = {'f1': 'D3 = D0 ^ D1 ^ D2',
                  'f2': 'D6 = D4 & D5'}
```

Once the expressions have been defined, they can be passed to the BooleanGenerator function.

```python
boolean_generator.setup(function_specs)
```

```python
boolean_generator.run()
```

To disconnect the IO pins, stop it.

```python
boolean_generator.stop()
```

If users want to use a different configuration, before calling `setup()` again, users have to call `reset()`; this will clear all the reservations on the currently used pins.

```python
boolean_generator.reset()
```

More information about the Boolean Generator module and its API can be found in the *pynq.lib.logictools Package* section.
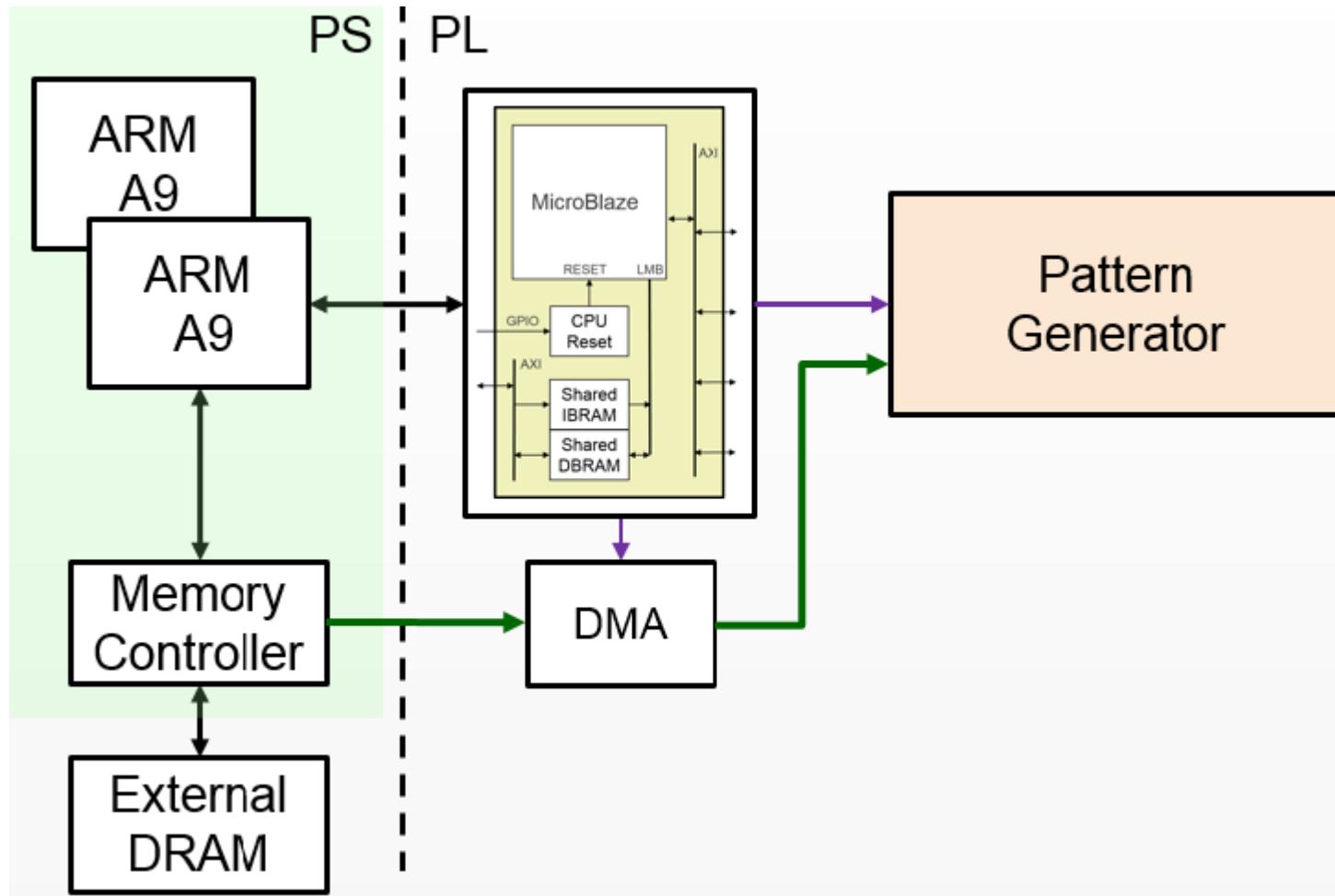
For more examples see the Logictools Notebooks folder on the Pynq-Z1 board in the following directory:

```
<Jupyter Home>/logictools/
```

## Pattern Generator

The Pattern Generator allows arbitrary digital patterns to be streamed to IO. This can be used to test or control external circuits or devices.

**Block Diagram**



The Pattern Generator supports up to 64K pattern words. Though the memory is 32-bits wide, only least significant 20 bits are used which are routed to the Arduino pins. A data word is generated once every rising edge of the sample clock.

The sample clock is programmable. The minimum sample clock speed is 252 KHz, and the maximum speed is 10 MHz.

The Pattern Generator class is instantiated by importing it from the logictools sub-package.

**Examples**

```python
from pynq.overlays.logictools import LogicToolsOverlay

logictools = LogicToolsOverlay('logictools.bit')

pattern_generator = logictools.pattern_generator
```

More information about the Pattern Generator module and its API can be found in the *pynq.lib.logictools Package* section.
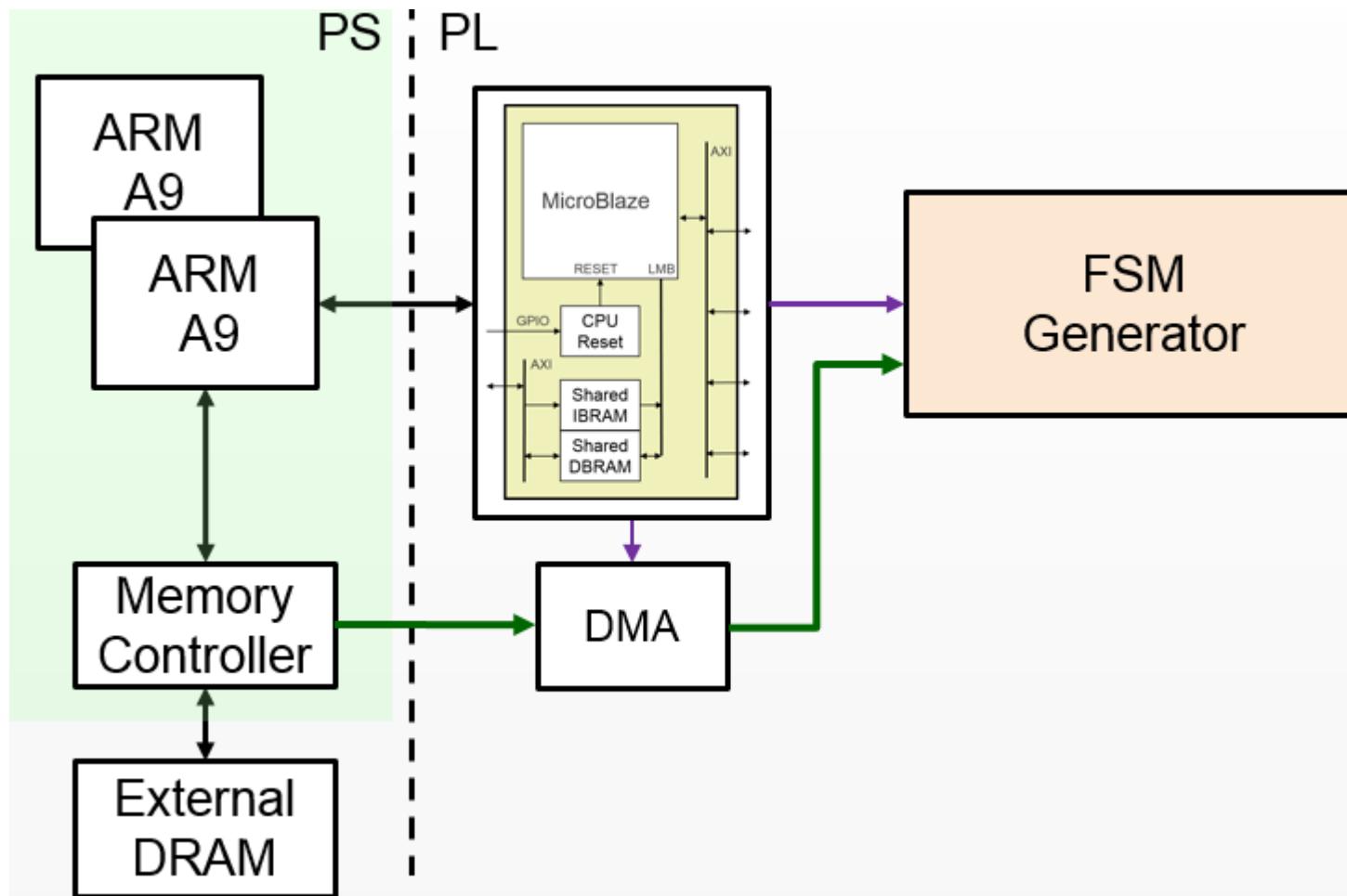
For more examples see the Logictools Notebooks folder on the Pynq-Z1 board in the following directory:

```
<Jupyter Home>/logictools/
```

### FSM Generator

The Finite State Machine (FSM) Generator can generate a finite state machine in programmable hardware from a Python description.

### Block Diagram



The FSM generator has an internal Block Memory which implements the finite state machine. The 20 pins on the Arduino shield header are available. The FSM must have a minimum of 1 input, allowing a maximum of 19 outputs. The maximum number of inputs is 8. For example, based on the number of inputs, the following configurations are available:

| # Inputs | Max # States | Max # Outputs |
|----------|--------------|---------------|
| 8        | 31           | 12            |
| 7        | 63           | 13            |
| 6        | 127          | 14            |
| 5        | 255          | 15            |
| 4        | 511          | 16            |

The Trace Analyzer is controlled by a MicroBlaze subsystem. It is connected to a DMA, also controlled by the MicroBlaze subsystem which is used to load configuration information, including the Block Memory configuration to implement the FSM.

The configuration for the FSM, Input pins, output pins, internal states, and state transitions, can be specified in a text format.

### Examples

```
fsm_spec = {'inputs': [('reset','D0'), ('direction','D1')],
            'outputs': [('bit2','D3'), ('bit1','D4'), ('bit0','D5')],
            'states': ['S0', 'S1', 'S2', 'S3', 'S4', 'S5', 'S6', 'S7'],
            'transitions': [['01', 'S0', 'S1', '000'],
                            ['00', 'S0', 'S7', '000'],
                            ['01', 'S1', 'S2', '001'],
                            ['00', 'S1', 'S0', '001'],
                            ['01', 'S2', 'S3', '011'],
                            ['00', 'S2', 'S1', '011'],
                            ['01', 'S3', 'S4', '010'],
                            ['00', 'S3', 'S2', '010'],
                            ['01', 'S4', 'S5', '110'],
                            ['00', 'S4', 'S3', '110'],
                            ['01', 'S5', 'S6', '111'],
                            ['00', 'S5', 'S4', '111'],
                            ['01', 'S6', 'S7', '101'],
                            ['00', 'S6', 'S5', '101'],
                            ['01', 'S7', 'S0', '100'],
                            ['00', 'S7', 'S6', '100'],
                            ['1-', '*',  'S0', '']]}
```
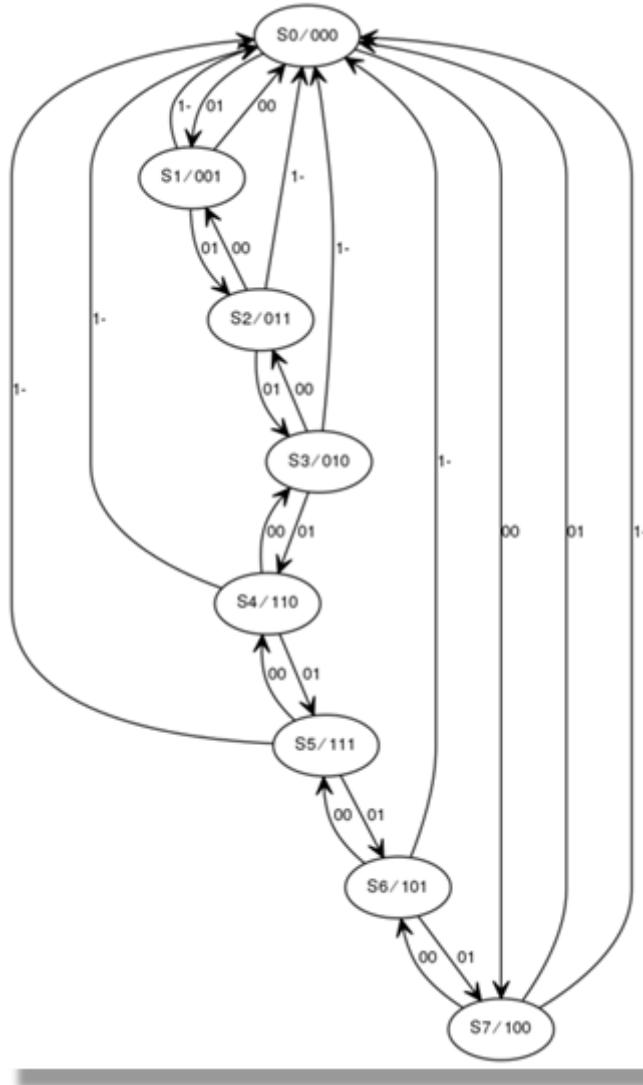
The FSM specification is passed to the `setup()`. The `run()` method can then be used to start the FSM.

The FSM Generator can be used in a similar way to the other generators.

Two additional methods are available to show the FSM state diagram in a notebook, and to display the waveform from the FSM.

```
show_state_diagram()
show_waveform()
```

Example of a state diagram:

More information about the FSM Generator module and its API can be found in the *pynq.lib.logictools Package* section.

For more examples see the Logictools Notebooks folder on the Pynq-Z1 board in the following directory:

```
<Jupyter Home>/logictools/
```

### Trace Analyzer
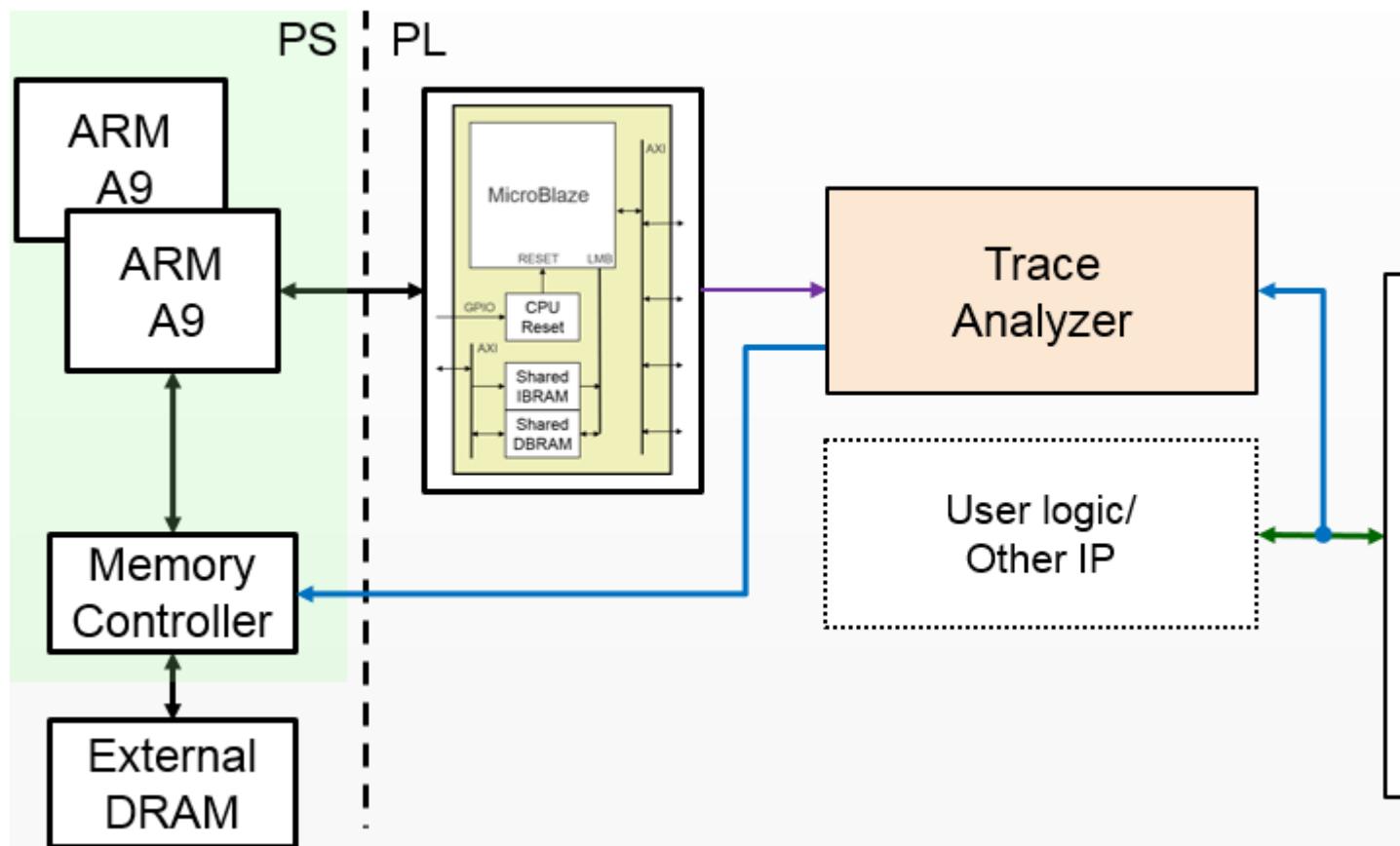
Traditional on-chip debug allows FPGA resources to be used to monitor internal or external signals in a design for debug. The debug circuitry taps into signals in a design under test, and saves the signal data as the system is operating. The debug data is saved to on-chip memory, and can be read out later for offline debug and analysis. One of the limitations of traditional on-chip debug is that amount of local memory usually available on chip is relatively small. This means only a limited amount of debug data can be captured (typically a few Kilobytes).

The on-chip debug concept has been extended to allow trace debug data to be saved to DDR memory. This allows more debug data to be captured. The data can then be analyzed using Python.

The trace analyzer monitors the external PL Input/Output Blocks (IOBs) on the PMOD and Arduino interfaces. The

IOBs are tri-state. This means three internal signals are associated with each pin; an input (I), and output (O) and a tri-state signal (T). The Tri-state signal controls whether the pin is being used as a input or output. The trace analyzer is connected to all 3 signals for each IOP (PMOD and Arduino).

### Block Diagram



This allows the trace analyzer to read the tri-state, determine if the IOB is in input, or output mode, and read the appropriate trace data.

### Examples

More information about the Trace Analyzer module and its API can be found in the *pynq.lib.logictools Package* section.

For more examples see the Logictools Notebooks folder on the Pynq-Z1 board in the following directory:

```
<Jupyter Home>/logictools/
```

### Video
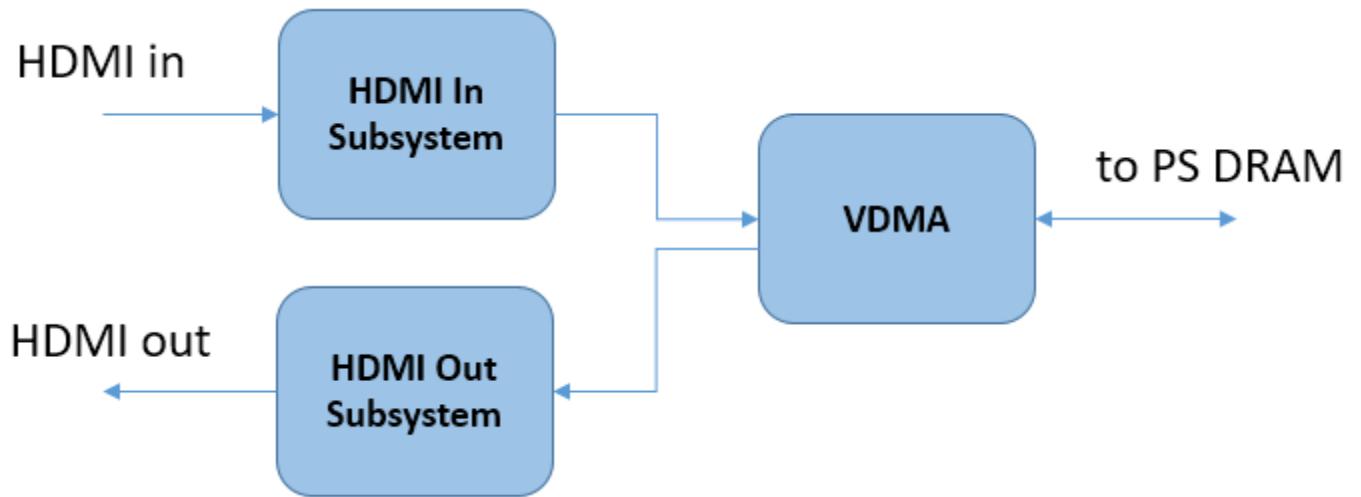
The Video subpackage contains a collection of drivers for reading from the HDMI-In port, writing to the HDMI-Out port, transferring data, setting interrupts and manipulating video frames.

The Video hardware subsystem consists of a HDMI-In block, a HDMI-Out block, and a Video DMA. The HDMI-In and HDMI-Out blocks also support color space conversions, e.g. from YCrCb to RGB and back, and changing the
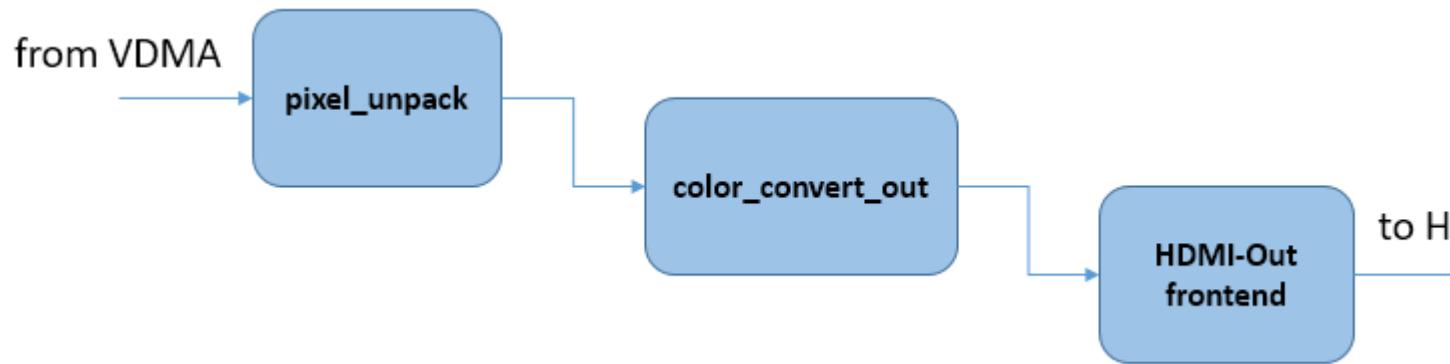
number of channels in each pixel.

Video data can be captured from the HDMI-In, and streamed to DRAM using the Video DMA, or streamed from DRAM to the HDMI-Out.

**Block Diagram**
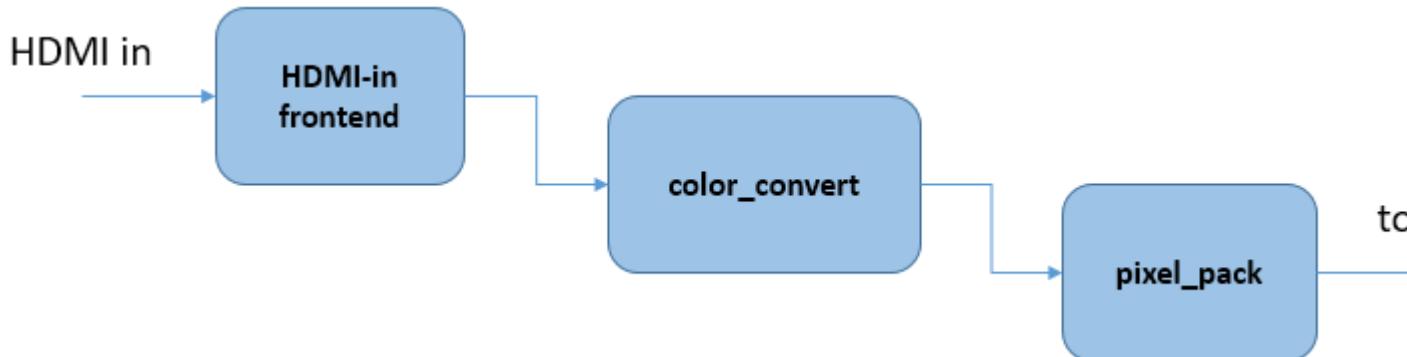


**HDMI-In**



The Pixel Unpack and the Color Convert block allow conversion between different color spaces at runtime.

**HDMI-Out**



The HDMI-Out is similar to HDMI-In. It has a Pixel Pack block (instead of the *Unpack* block for HDMI-In) and a Color Convert block.

**Video Front-ends**

The video library supports two different video front-ends. The front-end is responsible for converting the signals at the pins of the device into a 24-bit-per-pixel, BGR formatted AXI stream that can be used by the rest of the pipeline. For the Pynq-Z1 and Pynq-Z2 a DVI-based front-end provides for resolutions of up to 1080p although due to the speed ratings of the differential pins only up to 720p is officially supported. This front-end is a drop-in IP included in the PYNQ IP library. For the ZCU104 the Xilinx HDMI subsystems are used to support full 4k support however recreating this bitstream will require a license. The HDMI subsystems also require connecting to a HDMI PHY responsible for driving the transceivers. This code can be seen in the ZCU104's *base.py*. All custom overlays needing to use the video subsystem should use this setup code.

**Processing Options**

There are 3 main ways that some processing could be carried out on a Video pipeline.

1. Read, write and process frames in Python on the PS

2. Modify the overlay to insert an IP into the video pipeline

3. Modify the overlay to connect an IP to an available interface in the overlay

**Python Processing**

Input and output frames are represented as numpy arrays frames are read from the HDMI input and written to the HDMI output. The HDMI input will return the most recent complete frame received or block if necessary until the frame is ready. The HDMI output will continue outputting the last frame written until a new frame is provided.

As frames are numpy arrays they can be directly used in other Python libraries including the Python OpenCV libraries.

Examples of using the video API with OpenCV can be found in the video notebooks.

Processing video in the PS will be relatively slow. Where possible low level processing should be carried out in the PL. The video subsystem supports basic color space and pixel type conversions in hardware before passing a frame to the Python environment to improve performance of OpenCV and processing by other libraries.

### Pipeline Processing

The HDMI signal blocks are AXI-stream. A custom AXI-stream IP with an input stream and output stream could be inserted into the video pipeline, either on the HDMI-In side, or HDMI-Out side. However, usually IP to process the video stream will assume a fixed color space/pixel format and this will determine where an IP should be connected in the Video pipeline.

It is usually appropriate to insert the IP after the pixel_pack block on the HDMI-In block, or before the pixel_unpack block on the HDMI-Out side. This gives flexibility to use the video subsystem color space conversion blocks before and after the custom IP.

The video pipelines of the Pynq-Z1 and Pynq-Z2 boards run at 142 MHz with one pixel-per-clock, slightly below the 148.5 MHz pixel clock fo 1080p60 video but sufficient once blanking intervals are taken into account. for the ZCU104 board the pipeline runs at 300 MHz and two pixels-per-clock to support 4k60 (2160p) video.

### Batch Processing

An IP block can be added to an overlay and connected to an available interface. Usually a DMA would be used to stream the input frame buffer to the IP, and send the processed data back to the output frame buffer in DRAM.

Note that the DRAM is likely to be a bottleneck for video processing. The Video data is written to DRAM, then read from DRAM and send to the custom IP and is written back to DRAM, where it is read by the HDMI out.

For the Pynq-Z1 which has a 16-bit DRAM, up to 1080p cwgraysc (8-bits per pixel) can be processed at ~60fps alongside the framebuffer memory bandwidth, but this is very close to the total memory bandwidth of the system. The ZCU104 with its much larger memory bandwidth can support 4k video at 60 FPS at 24-bit colour.

### Examples

More information about the Video subpackage, its components, and its their APIs can be found in the *pynq.lib.video Module* section.

For more examples, see the Video Notebooks folder on the Pynq-Z1, Pynq-Z2 or ZCU104 board in the following directory:

```
<Jupyter Home>/base/video
```

### Initialization

Set up an instance of the HDMI-in, and HDMI-out.

```python
from pynq import Overlay
from pynq.lib.video import *

base = Overlay('base.bit')
hdmi_in = base.video.hdmi_in
hdmi_out = base.video.hdmi_out
```

### Configuration

The HDMI-in interface is enabled using the `configure` function which can optionally take a colorspace parameter. If no colorspace is specified then 24-bit BGR is used by default. The HDMI-in *mode* can be used to configure the

HDMI-out block. This specifies the output color space and resolution.

```
hdmi_in.configure()
hdmi_out.configure(hdmi_in.mode)
```

### Execution

Once the HDMI controllers have been configured, they can be started:

```
hdmi_in.start()
hdmi_out.start()
```

To connect a simple stream from HDMI-in to HDMI-out, the two streams can be tied together.

```
hdmi_in.tie(hdmi_out)
```

This takes the unmodified input stream and passes it directly to the output. While the input and output are tied frames can still be read from the input but any call to `hdmi_out.writeframe` will end the tie.

```
frame = hdmi_in.readframe()
...
hdmi_out.writeframe(frame)
```

This would allow some processing to be carried out on the HDMI-in *frame* before writing it to the HDMI-out.

### Color Space Conversion

The video subsystem supports general color space conversions so that frames in DRAM are in a format appropriate for any subsequent processing. The default color space is BGR(24-bit) with RGB (24-bit), RGBA (32-bit), BGR (24-bit), YCbCr (24-bit), and grayscale (8-bit) provided as built-in options.

The colorspace converter operates on each pixel independently using a 3x4 matrix to transform the pixels. The converter is programmed with a list of twelve coefficients in the following order:

| Channel | in1 | in2 | in3 | 1 |
|---------|-----|-----|-----|-----|
| out1 | c1 | c2 | c3 | c10 |
| out2 | c4 | c5 | c6 | c11 |
| out3 | c7 | c8 | c9 | c12 |

Each coefficient should be a floating point number between -2 and +2.

The pixels to and from the HDMI frontends are in BGR order so a list of coefficients to convert from the input format to RGB would be:

```
[0, 0, 1,
 0, 1, 0,
 1, 0, 0,
 0, 0, 0]
```

reversing the order of the channels and not adding any bias.

The driver for the colorspace converters has a single property that contains the list of coefficients.

```
colorspace_in = base.video.hdmi_in.color_convert
colorspace_out = base.video.hdmi_out.color_convert

bgr2rgb = [0, 0, 1,
           0, 1, 0,
           1, 0, 0,
           0, 0, 0]

colorspace_in.colorspace = bgr2rgb
colorspace_out.colorspace = bgr2rgb
```

### Pixel Format

The default pixel format for the HDMI frontends is 24-bit - that is three 8-bit channels. This can be converted to 8, 16, 24 or 32 bits.

8-bit mode selects the first channel in the pixel (and drops the other two) 16-bit mode can either select the first two channels or select the first and performs chroma resampling of the other two resulting in 4:2:2 formatted frames. 24-bit mode is pass-through, and doesn't change the format 32-bit mode pads the stream with additional 8-bits.

```
pixel_in = base.video.hdmi_in.pixel_pack
pixel_out = base.video.hdmi_out.pixel_unpack

pixel_in.bits_per_pixel = 8
pixel_out.bits_per_pixel = 16
pixel_out.resample = False
```

### Video Pipeline

As the `hdmi_in.readframe` and `hdmi_out.writeframe` functions may potentially block if a complete frame has not yet been read or written, `_async` versions of these functions also exist. One use for the asynchronous versions is if frames are being transferred to a separate accelerator using a DMA engine. The DMA driver is also asyncio aware so the computation can be written as two tasks. One to retrieve frames from the Video DMA and forward them to the accelerator and a second task to bring frames back from the accelerator.

```python
async def readframes():
    while True:
        frame = await hdmi_in.readframe_async()
        dma.sendchannel.transfer(frame)
        await dma.sendchannel.wait_async()
        frame.freebuffer()

async def writeframes():
    while True:
        frame = hdmi_out.newframe()
        dma.recvchannel.transfer(frame)
        await dma.recvchannel.wait()
        await hdmi_out.writeframe_async(frame)
```

### Zynq Ultrascale+ DisplayPort
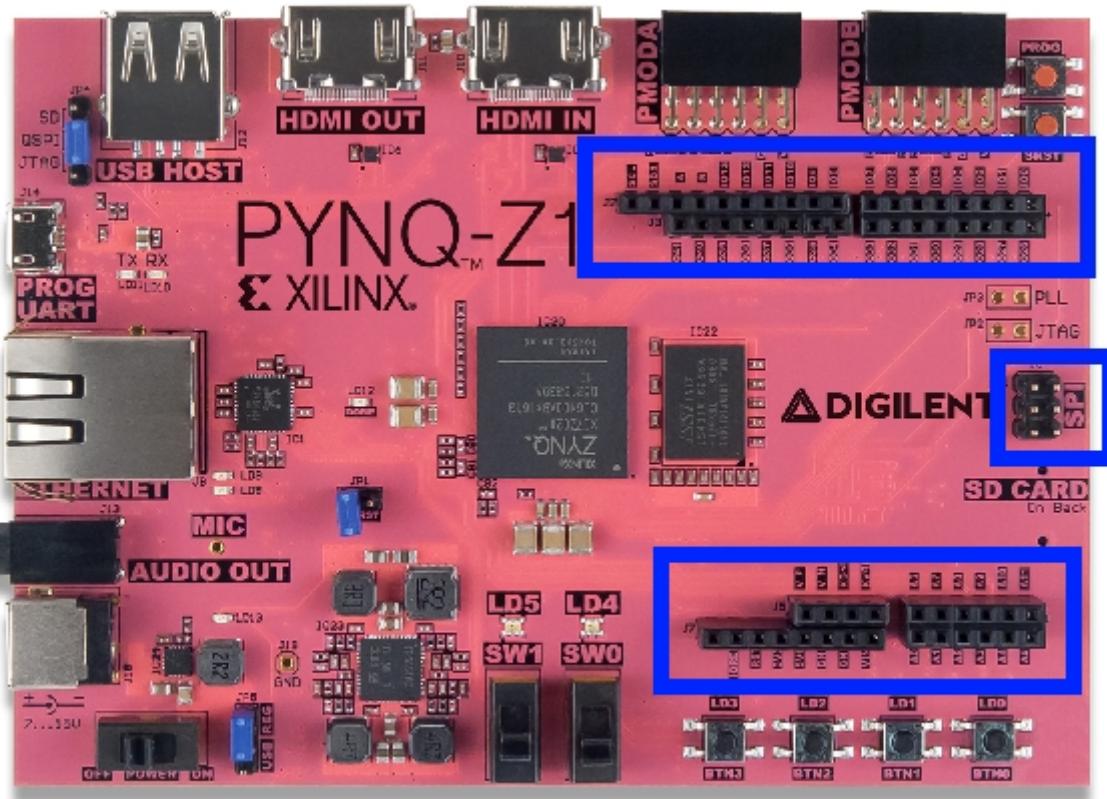
On Zynq Ultrascale+ devices there is a hardened DisplayPort interface that may be exposed on the board. On all supported boards the PYNQ environment will bring up a Fluxbox-based desktop environment with the Chromium browser to allow easy access to Jupyter directly on the board. For high-performance video output over DisplayPort the PYNQ environment offers a `DisplayPort` class that offers a similar API to the HDMI output. The only change between the DisplayPort and HDMI outputs is that the colourspace cannot be changed dynamically and frames are not interoperable between the two subsystems. While PYNQ is using the DisplayPort output it will replace the desktop environment.

## 2.5.2 IOPs

### Arduino
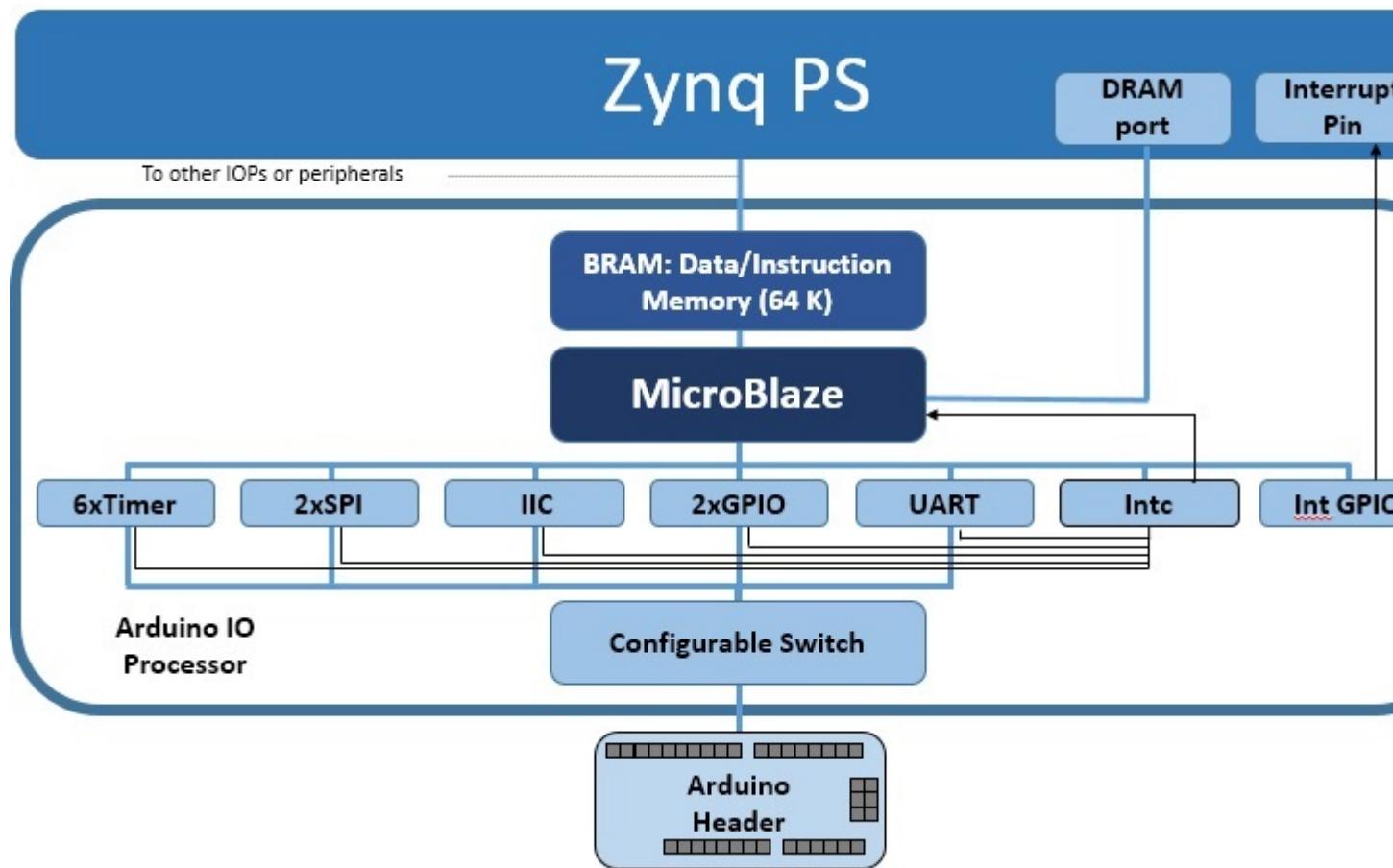
The Arduino subpackage is a collection of drivers for controlling peripherals attached to a Arduino port.

An Arduino connector can be used to connect to Arduino compatible shields to PL pins. Remember that appropriate controllers must be implemented in an overlay and connected to the corresponding pins before a shield can be used. Arduino pins can also be used as general purpose pins to connect to custom hardware using wires.



### Block Diagram

An Arduino PYNQ MicroBlaze is available to control the Arduino interface, if provided. The Arduino PYNQ MicroBlaze is similar to the Pmod PYNQ MicroBlaze, with more AXI Controllers.

---

As indicated in the diagram, the Arduino PYNQ MicroBlaze has a PYNQ MicroBlaze Subsystem, a configurable switch, and the following AXI controllers:

- AXI I2C

    - Frequency: 100KHz

    - Address mode: 7 bit

- 2x AXI SPI

    - Master mode

    - Transaction Width: 8

    - SCK Frequency: 6.25 MHz

    - FIFO Depth: 16

**Note:** One SPI controller is connected to the Arduino interface dedicated SPI pins.

- 3x AXI GPIO

    - 16 Input/Output pins total

- 6x AXI Timer

    - 32 bits

    - 1 Generate Output

- – 1 PWM Output
- 1x AXI UART
  - – 9600 Baud
- 1x AXI XADC
  - – 1V peak-to-peak *

> **Warning:** Analog inputs are supported via the internal Xilinx XADC. This supports inputs of 1V peak-to-peak. Note that the Arduino interface supports 0-5V analog inputs which is not supported by Zynq without external circuitry.

- AXI Interrupt controller

  Manages the interrupts of peripherals in the MicroBlaze subsystem.

- Interrupt GPIO

  An additional AXI GPIO is used to signal interrupt requests to the PS

- Configurable Switch

  Allows routing of signals from dedicated peripherals to the external interface.

| Peripheral | Pins |
|------------|------|
| UART | D0, D1 |
| I2C | SCL, SDA |
| SPI* | D10 - D13 |
| PWM | D3, D5, D6, D9, D10, D11 |
| Timer | D3 - D6 and D8 - D11 |

### Examples

In the *Base Overlay*, one Arduino PYNQ MicroBlaze instance is available. After the overlay is loaded this instance can be accessed as follows:

```python
from pynq.overlays.base import BaseOverlay
from pynq.lib.arduino import Arduino_LCD18

lcd = Arduino_LCD18(base.ARDUINO)
lcd.clear()
```

More information about the Arduino subpackage, its components, and its API can be found in the *pynq.lib.arduino Package* section.

For more examples, see the notebooks in the following directory on the PYNQ-Z1 board:

```
<Jupyter Dashboard>/base/arduino/
```

### Grove

The Grove peripherals can be accessed on Pmod pins using the *PYNQ Grove Adapter* and on Arduino pins using the *PYNQ Shield.*
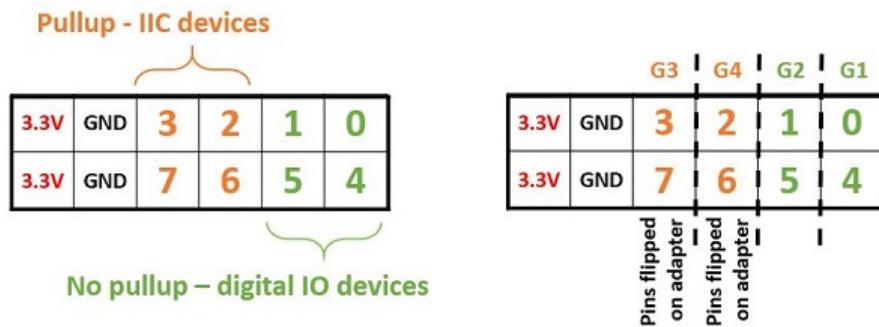
**Block Diagram**

**Pmod**

The first option for connecting Grove peripherals uses the Pmod PYNQ MicroBlaze. Grove devices can be connected to a board through the Pmod ports using the *PYNQ Grove Adapter*.
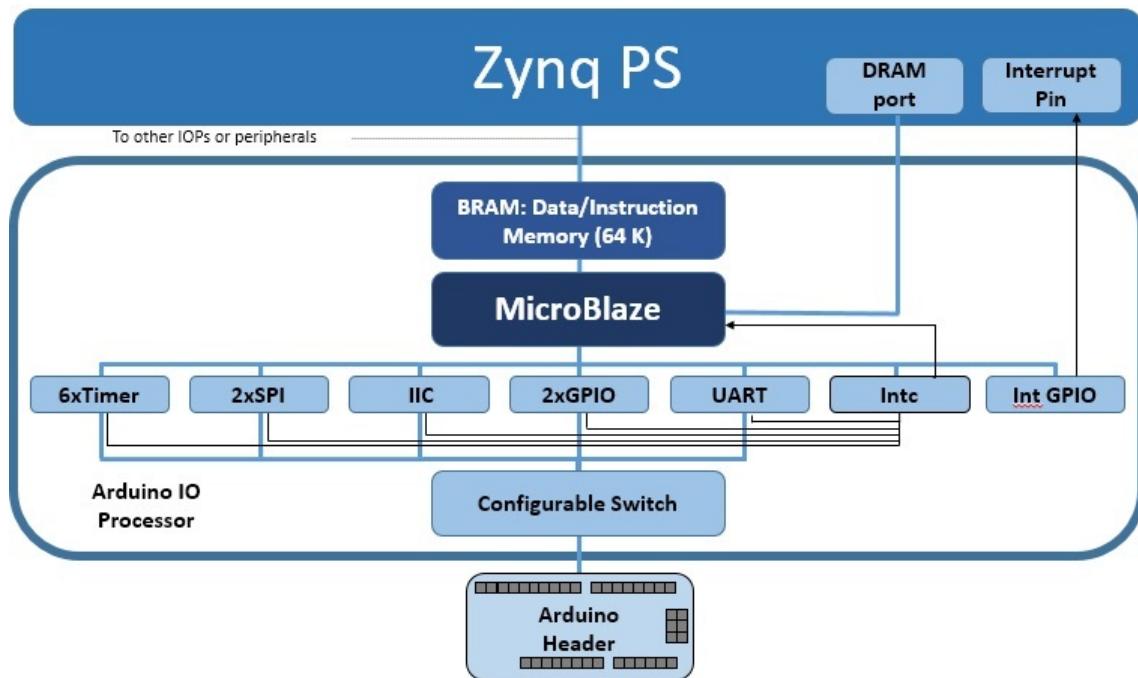


On the *PYNQ Grove Adapter* G1 and G2 map to Pmod pins [0,4] and [1,5], which are connected to pins with pull-down resistors. Ports G1 and G2 support the SPI protocol, GPIO, and timer Grove peripherals, but not IIC peripherals. Ports G3 and G4 map to pins [2,6], [3,7], which are connected to pins with pull-up resistors and support the IIC protocol and GPIO peripherals.
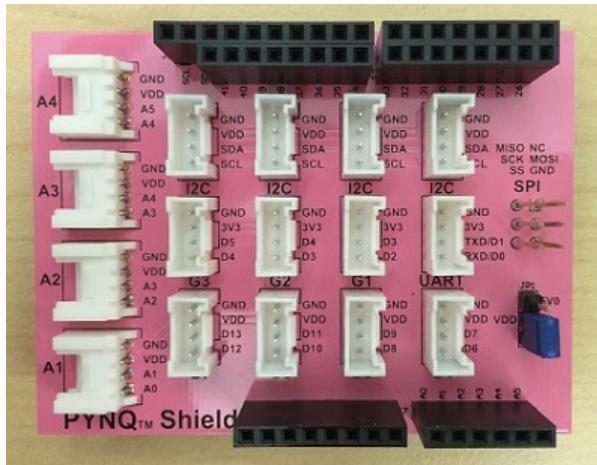


| Peripheral | Grove Port |
|------------|------------|
| I2C | G3, G4 |
| SPI | G1, G2 |

**Arduino**

The second option for connecting Grove peripherals uses the Arduino PYNQ MicroBlaze.

Grove devices can be connected to a board through the Arduino interface using the *PYNQ Shield*.



On the *PYNQ Shield* there are 4 IIC Grove connectors (labeled I2C), 8 vertical Grove Connectors (labeled G1-G7 and UART), and four horizontal Grove Connectors (labeled A1-A4). The SCL and SDA pins are connected to the SCL and SDA pins on the Arduino header.

The following table maps Grove Ports to communication protocols.

| Peripheral | Grove Port |
|---|---|
| UART | UART |
| I2C | A4, I2C (x4) |
| SPI | G7, G6 |
| GPIO | UART, G1 - G7 |

A list of drivers provided for Grove peripherals can be found in *pynq.lib.pmod Package* for the *PYNQ Grove Adapter* and in *pynq.lib.arduino Package* for the *PYNQ Shield*.

### Examples

In *Base Overlay*, two Pmod instances are available: PMODA and PMODB. After the overlay is loaded, the Grove peripherals can be accessed as follows:

```python
from pynq.overlays.base import BaseOverlay
from pynq.lib.pmod import Grove_Buzzer
from pynq.lib.pmod import PMOD_GROVE_G1

base = BaseOverlay("base.bit")

grove_buzzer = Grove_Buzzer(base.PMODB,PMOD_GROVE_G1)
grove_buzzer.play_melody()
```

More information about the Grove drivers in the Pmod subpackage, the supported peripherals, and APIs can be found in *pynq.lib.pmod Package*.

For more examples using the *PYNQ Grove Adapter*, see the notebooks in the following directory on the board:

```
<Jupyter Dashboard>/base/pmod/
```

In *Base Overlay*, one Arduino PYNQ MicroBlaze instance is available. After the overlay is loaded, the Grove peripherals can be accessed as follows:

```python
from pynq.overlays.base import BaseOverlay
from pynq.lib.arduino import Grove_LEDbar
from pynq.lib.arduino import ARDUINO_GROVE_G4

base = BaseOverlay("base.bit")

ledbar = Grove_LEDbar(base.ARDUINO,ARDUINO_GROVE_G4)
ledbar.reset()
```

More information about the Grove drivers in the Arduino subpackage, the supported peripherals, and APIs can be found in *pynq.lib.arduino Package*.

For more examples using the *PYNQ Shield*, see the notebooks in the following directory on the board:

```
<Jupyter Dashboard>/base/arduino/
```

### Pmod
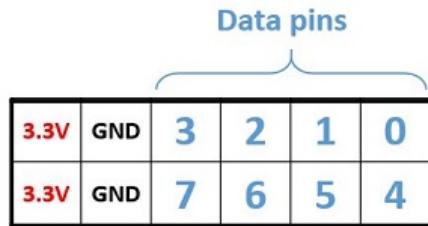
The Pmod subpackage is a collection of drivers for controlling peripherals attached to a Pmod port.

A Pmod port is an open 12-pin interface that is supported by a range of Pmod peripherals from Digilent and third party manufacturers. Typical Pmod peripherals include sensors (voltage, light, temperature), communication interfaces (Ethernet, serial, WiFi, Bluetooth), and input and output interfaces (buttons, switches, LEDs).
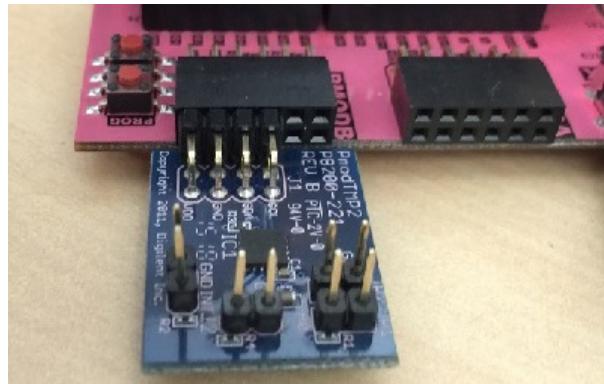


Each Pmod connector has 12 pins arranged in 2 rows of 6 pins. Each row has 3.3V (VCC), ground (GND) and 4 data pins. Using both rows gives 8 data pins in total.

Pmods come in different configurations depending on the number of data pins required. E.g. Full single row: 1x6 pins; full double row: 2x6 pins; and partially populated: 2x4 pins.



Pmods that use both rows (e.g. 2x4 pins, 2x6 pins), should usually be aligned to the left of the connector (to align with VCC and GND).
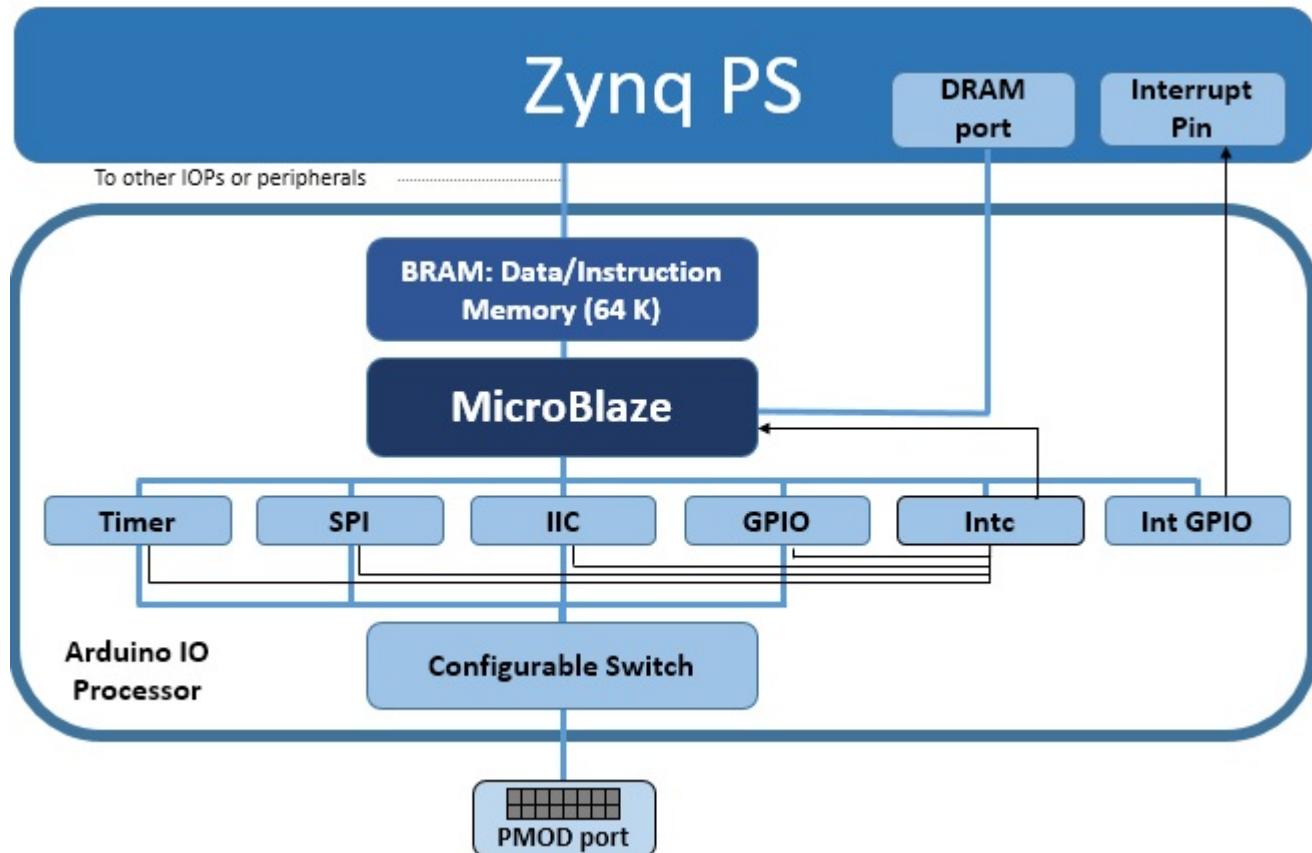


Pmod peripherals with only a single row of pins can be connected to either the top row or the bottom row of a Pmod port (again, aligned to VCC/GND). If you are using an existing driver/overlay, you will need to check which pins/rows are supported for a given overlay, as not all options may be implemented. e.g. the Pmod ALS is currently only supported on the top row of a Pmod port, not the bottom row.

All pins operate at 3.3V. Due to different pull-up/pull-down I/O requirements for different peripherals (e.g. IIC requires pull-up, and SPI requires pull-down) the Pmod data pins have different IO standards.

Pins 0,1 and 4,5 are connected to pins with pull-down resistors. This can support the SPI interface, and most peripherals. Pins 2,3 and 6,7 are connected to pins with pull-up resistors. This can support the IIC interface.

Pmods already take this pull up/down convention into account in their pin layout, so no special attention is required when using Pmods.

**Block Diagram**



As indicated in the diagram, each Pmod PYNQ MicroBlaze has a *MicroBlaze Subsystem* a configurable switch, and the following AXI controllers:

- AXI I2C

    - SCL Frequency 100 KHz

    - Address Mode: 7 bits

- AXI SPI

    - Master mode

    - Transaction Width: 8

    - SCK Frequency: 6.25 MHz

    - FIFO Depth: 16

- AXI GPIO

    - 8 Input/Output pins

- AXI Timer

    - 32 bits

    - 1 Generate Output

    - 1 PWM Output

- AXI Interrupt controller

  Manages the interrupts of peripherals in the MicroBlaze subsystem.

- Interrupt GPIO

  - An additional AXI GPIO is used to signal interrupt requests to the PS

- Configurable Switch

  - Allows routing of signals from dedicated peripherals to the external interface.

A list of drivers provided for Pmod peripherals can be found in the *pynq.lib.pmod Package* section.

### Examples

In the *Base Overlay*, two Pmod instances are available: PMODA and PMODB. After the overlay is loaded theses instances can be accessed as follows:

```python
from pynq.overlays.base import BaseOverlay
from pynq.lib import Pmod_Timer

base = BaseOverlay("base.bit")

pt = Pmod_Timer(base.PMODA,0)
pt.stop()
```

More information about the Pmod subpackage, its components, and its API can be found in the *pynq.lib.pmod Package* section.
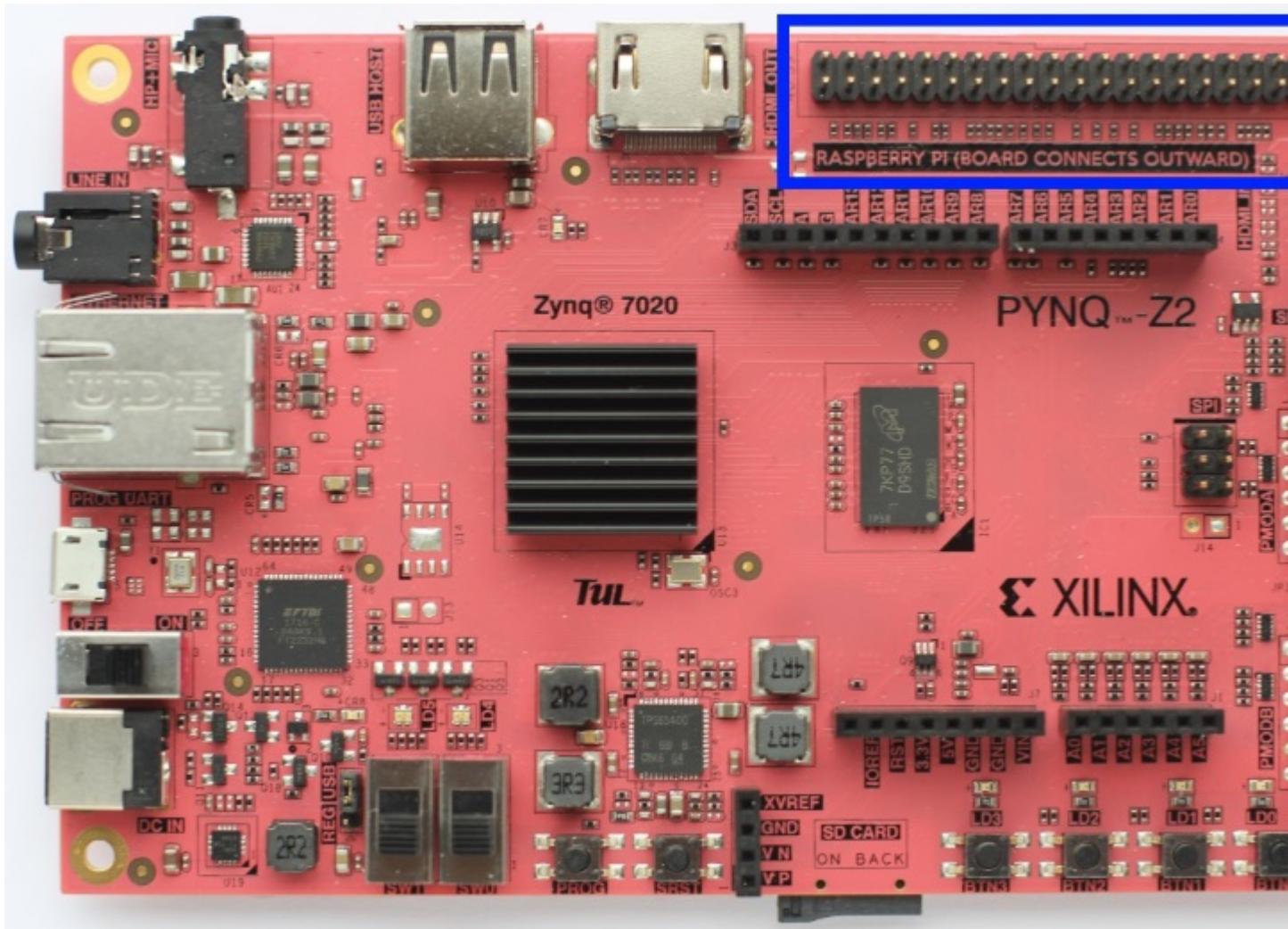
For more examples, see the notebooks in the following directory on the board:

```
<Jupyter Dashboard>/base/pmod/
```

### RPi
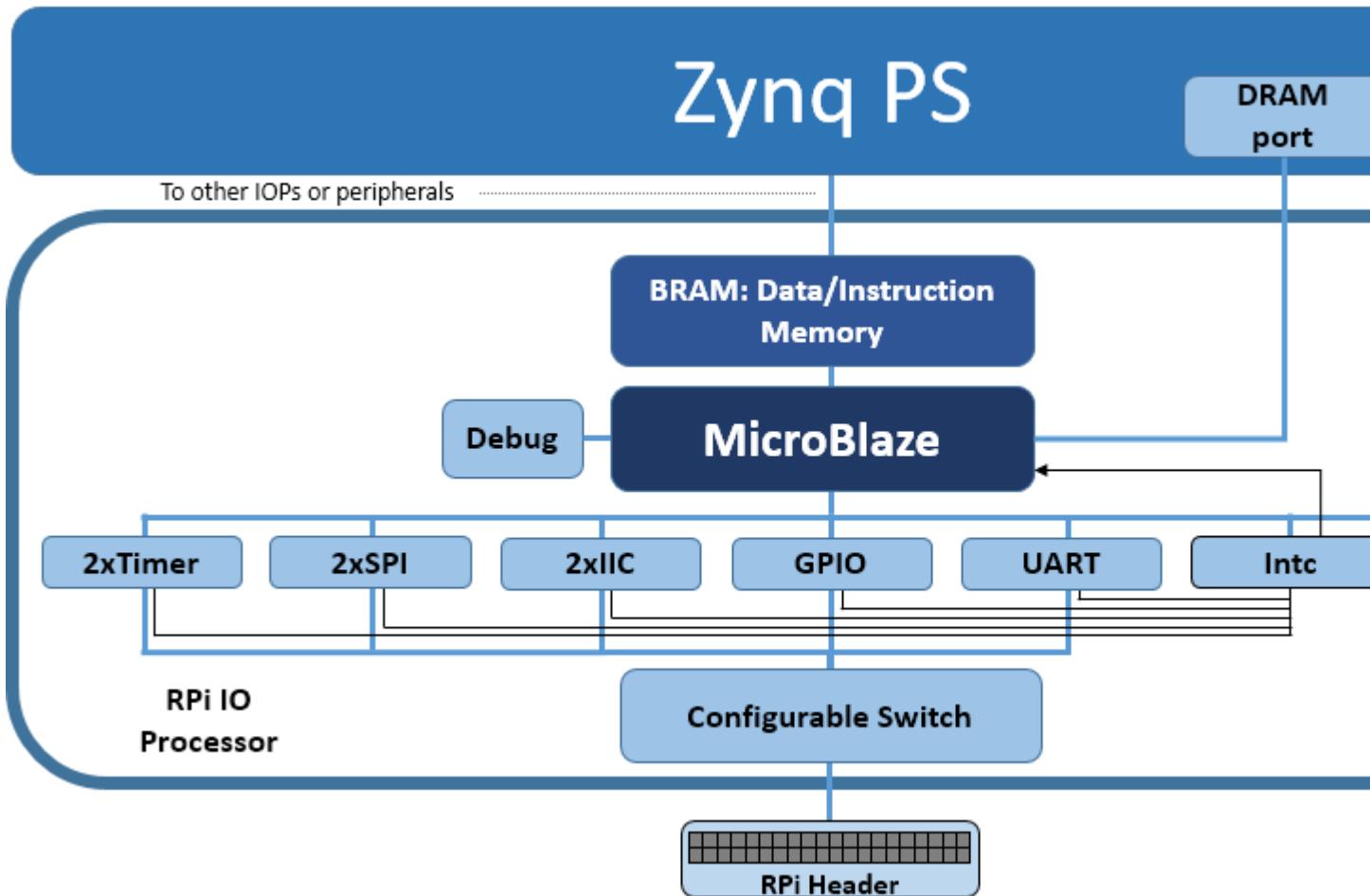
The *rpi* subpackage is a collection of drivers for controlling peripherals attached to a RPi (Raspberry Pi) interface.

The RPi connector can be used to connect to Raspberry Pi compatible peripherals to PL pins. Remember that appropriate controllers must be implemented in an overlay and connected to the corresponding pins before a shield can be used. The RPi pins can also be used as general purpose pins to connect to custom hardware using wires.

### Block Diagram

The RPi PYNQ MicroBlaze is available to control the RPi interface.

As indicated in the diagram, the RPi PYNQ MicroBlaze has a PYNQ MicroBlaze Subsystem, a configurable switch, and the following AXI controllers:

- 2x AXI I2C

    - Frequency: 100KHz

    - Address mode: 7 bit

- 2x AXI SPI

    - Master mode

    - Transaction Width: 8

    - SCK Frequency: 6.25 MHz

    - FIFO Depth: 16

---

**Note:** One SPI controller is connected to the Arduino interface dedicated SPI pins.

---

- 1x AXI GPIO

    - 28 Input/Output pins total

- 2x AXI Timer

    - 32 bits

- 1 Generate Output

- 1 PWM Output

- 1x AXI UART

  - 115200 Baud

- AXI Interrupt controller

  Manages the interrupts of peripherals in the MicroBlaze subsystem.

- Interrupt GPIO

  An additional AXI GPIO is used to signal interrupt requests to the PS

- Configurable Switch

  Allows routing of signals from dedicated peripherals to the external interface.

More information about the RPi subpackage, its components, and its API can be found in the *pynq.lib.rpi Package* section.

For more examples, see the notebooks in the following directory on the PYNQ-Z2 board:

```
<Jupyter Dashboard>/base/rpi/
```

### 2.5.3 PynqMicroBlaze

**MicroBlaze Subsystem**

The PYNQ MicroBlaze subsystem can be controlled by the PynqMicroblaze class. This allows loading of programs from Python, controlling executing by triggering the processor reset signal, reading and writing to shared data memory, and managing interrupts received from the subsystem.

Each PYNQ MicroBlaze subsystem is contained within an IO Processor (IOP). An IOP defines a set of communication and behavioral controllers that are controlled by Python. There are currently three IOPs provided with PYNQ: Arduino, PMOD, and Logictools.

**Block Diagram**
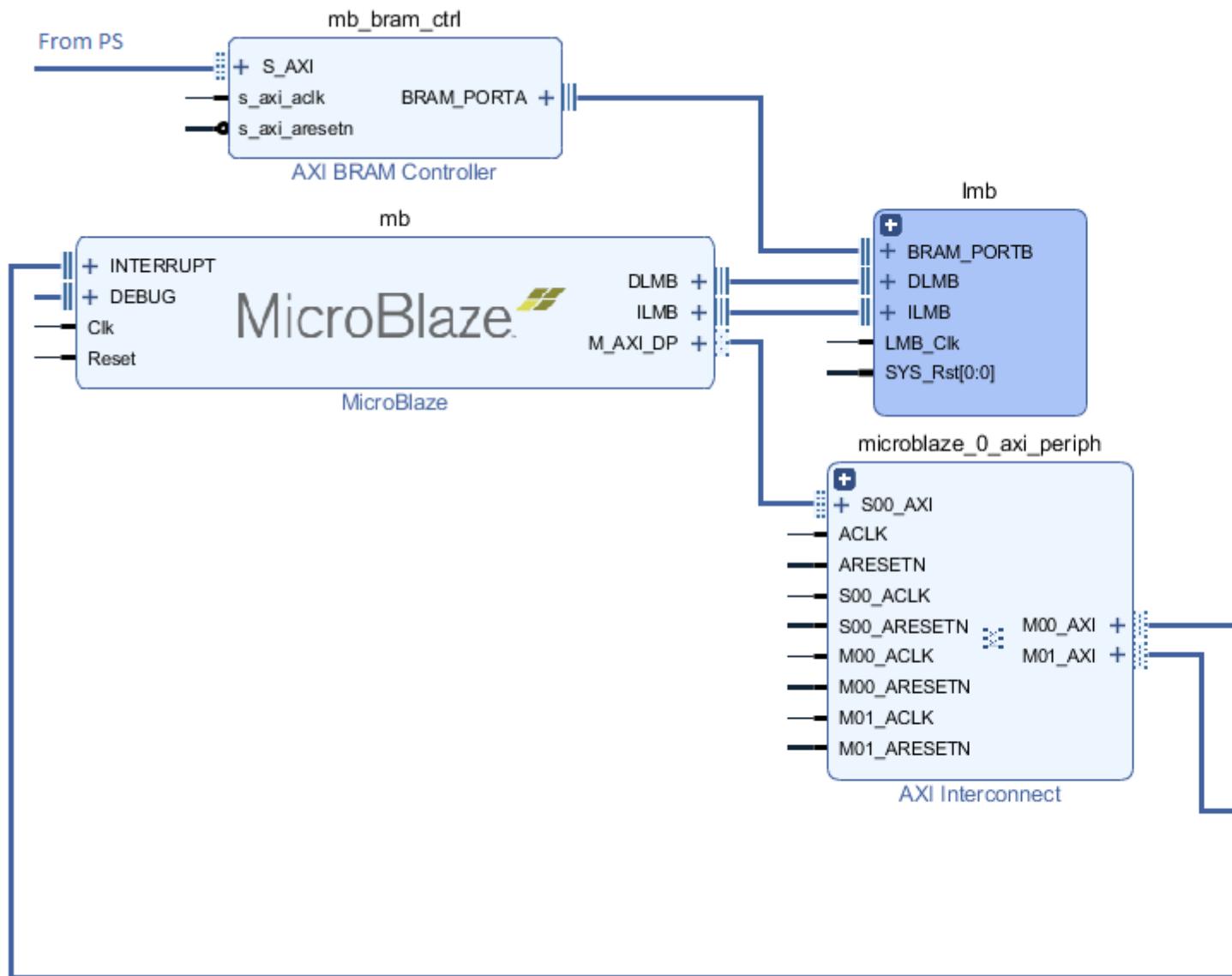
A PYNQ MicroBlaze subsystem consists of a MicroBlaze processor, AXI interconnect, Interrupt controller, an Interrupt Requester, and External System Interface, and Block RAM and memory controllers.

The AXI interconnect connects the MicroBlaze to the interrupt controller, interrupt requester, and external interface.

- The Interrupt Controller is the interface for other communication or behavioral controllers connected to the MicroBlaze Processor.

- The Interrupt Requester sends interrupt requests to the Zynq Processing System.

- The External Interface allows the MicroBlaze subsystem to communicate with other communication, behavioral controllers, or DDR Memory.

- The Block RAM holds MicroBlaze Instructions and Data.

The Block RAM is dual-ported: One port connected to the MicroBlaze Instruction and Data ports; The other port is connected to the ARM® Cortex®-A9 processor for communication.

If the External Interface is connected to DDR Memory, DDR can be used to transfer large data segments between the PS (Python) and the Subsystem.

### Examples

In the *Base Overlay*, three IOP instances with PYNQ Microblaze Subsystems are available: iop1 (PMODA), iop2 (PMODB), and iop3 (Arduino). After the overlay is loaded these can be accessed as follows:

```python
from pynq.overlays.base import BaseOverlay
from pynq.lib import PynqMicroblaze

base = BaseOverlay('base.bit')

mb = PynqMicroblaze(base.iop1.mb_info,
                    "/home/xilinx/pynq/lib/pmod/pmod_timer.bin")
mb.reset()
```

More information about the PynqMicroblaze class, and its API can be found in the *pynq.lib.pynqmicroblaze.pynqmicroblaze Module* section.

*Pmod*, *Arduino*, and *Grove* classes are subclasses of the PynqMicroBlaze class, and further example notebooks can be found in those sections.

### Creating a New PYNQ Microblaze

Any hierarchy that contains a Microblaze and meets the above requirements of having AXI-accessible code memory, a PS interrupt line and a reset line can be used in PYNQ. However in order to use your Microblaze with the IPython magic you must provide a board support package (BSP). BSPs are generated from Xilinx SDK and contain all of the drivers and configuration data for the peripherals attached to the Microblaze.

PYNQ provides a TCL script to generate the BSP from the hardware description file which can be found in the `boards/sw_repo` directory of the repository, along with the drivers needed for Python/C communication and the `pynqmb` hardware abstraction library. Creating and using the BSP requires the following steps:

1. Export Hardware from Vivado to generate the HDF file

2. In the `boards/sw_repo` directory run `make HDF=$HDF_FILE`. If no HDF is provided then the Base Overlay BSPs will be generated.

3. Copy the generated BSP on to the board and ensure it is name `bsp_${hierarchy}` where `${hierarchy}` is the name of the Microblaze hierarchy in your design. If you have multiple Microblazes it can all be a prefix common to all hierarchies.

4. Call `add_bsp(BSP_DIR)` in Python. This can be done as part of initialisation of your overlay.

These steps will integrate the BSP into the PYNQ runtime and allow the IPython `%%microblaze` magic to be used with the new Microblaze subsystem.

If you wish to reuse an existing subsystem, as long as it is named in accordance with the Base Overlay - e.g. `iop_pmod*` or `iop_arduino*` then the correct BSP will be used automatically.

### Microblaze RPC

The PYNQ Microblaze infrastructure is built on top of a remote procedure call (RPC) layer which is responsible for forwarding function calls from the Python environment to the Microblaze and handling all data transfer.

### Supported Function Signatures

The RPC layer supports a subset of the C programming language for interface functions, although any functions may be used internally within the Microblaze program. Any function which does not conform to these requirements will be ignored. The limitations are:

1. No `struct` or `union` for either parameters or return types.

2. No returning of pointer types

3. No pointers to pointers

### Data Transfer

All return values are passed back to Python through copying. The transfer of function arguments depends on the type used. For a given non-void primitive the following semantics:

- Non-pointer types are copied from PYNQ to the microblaze

- Const pointer types are copied from Python to the Microblaze

---

- Non-const pointer types are copied from Python to the Microblaze and then copied back after completion of the function.

The timeline of the execution of the function can be seen below:

The Python `struct` module is used to convert the Python type passed to the function into the appropriately sized integer or floating point value for the Microblaze. Out of range values will result in an exception being raised and the Microblaze function not running. Arrays of types are treated similarly with the `struct` module used to perform the conversion from an array of Python types to the C array. For non-const arrays, the array is updated in place so that the return values are available to the caller. The only exception to these conversion rules are `char` and `const char` pointers which are optimised for Python `bytearray` and `bytes` types. Note that calling a function with a non-const `char*` argument with a `bytes` object will result in an error because `bytes` objects are read-only. This will caught prior to the Microblaze function being called.

### Long-running Functions

For non-void return functions, the Python functions are synchronous and will wait for the C function to finish prior to returning to the caller. For functions that return void then the function is called asynchronously and the Python function will return immediately. This entails long-running, independent functions to run on the Microblaze without blocking the Python thread. While the function is running, no other functions can be called unless the long-running process frequently calls `yield` (from `yield.h`) to allow the RPC runtime to service requests. Please note - there is no multi-threading available inside the Microblaze so attempting to run two long-running processes simultaneously will result in only one executing regardless of the use of `yield`.

### Typedefs

The RPC engine fully supports typedefs and provides an additional mechanism to allow for C functions to appear more like Python classes. The RPC layer recognises the idiom where the name of a typedef is used as the prefix for a set of function names. Taking an example from the PYNQ Microblaze library, the `i2c` typedef has corresponding functions `i2c_read` and `i2c_write` which take an `i2c` type as the first parameter. With this idiom the RPC creates a new class called `i2c` which has `read` and `write` methods. Any C functions returning an `i2c` typedef now return an instance of this class. For this conversion to be done, the following three properties must hold:

1. The typedef is of a primitive type
2. There is at least one function returning the typedef
3. There is at least one function named according to the pattern

### Microblaze Library

The PYNQ Microblaze library is the primary way of interacting with Microblaze subsystems. It consists of a set of wrapper drivers for I/O controllers and is optimised for the situation where these are connected to a PYNQ I/O switch.

This document describes all of the C functions and types provided by the API - see the Python/C interoperability guide for more details on how this API translates into Python.

### General Principles

This library provides GPIO, I2C, SPI, PWM/Timer and UART functionality. All of these libraries follow the same design. Each defines a type which represents a handle to the device. `*_open` functions are used in situations where there is an I/O switch in the design and takes a set of pins to connect the device to. The number of pins depends on the

protocol. `*_open_device` opens a specific device and can be passed either the base address of the controller or the index as defined by the BSP. `*_close` is used to release a handle.

### GPIO Devices

GPIO devices allow for one or multiple pins to be read and written directly. All of these functions are in `gpio.h`

#### `gpio` type

A handle to one or more pins which can be set simultaneously.

#### `gpio gpio_open(int pin)`

Returns a new handle to a GPIO device for a specific pin on the I/O switch. This function can only be called if there is an I/O switch in the design.

#### `gpio gpio_open_device(unsigned int device)`

Returns a handle to an AXI GPIO controller based either on the base address or device index. The handle will allow for all pins on channel 1 to be set simultaneously.

#### `gpio gpio_configure(gpio parent, int low, int hi, int channel)`

Returns a new handle tied to the specified pins of the controller. This function does not change the configuration of the parent handle.

#### `void gpio_set_direction(gpio device, int direction)`

Sets the direction of all pins tied to the specified handle. The direction can either be `GPIO_IN` or `GPIO_OUT`.

#### `void gpio_write(gpio device, unsigned int value)`

Sets the value of the output pins represented by the handle. If the handle represents multiple pins then the least significant bit refers to the lowest index pin. Writing to pins configured as input has no effect.

#### `unsigned int gpio_read(gpio device)`

Reads the value of input pins represented by the handle, If the handle represents multiple pins then the least significant bit refers to the lowest index pin. Read from pins configured as output results in 0 being returned.

#### `void gpio_close(gpio_device)`

Returns the specified pins to high-impedance output and closes the device.

### I2C Devices

The I2C driver is designed for master operation only and provides interfaces to read and write from a slave device. All of these functions are in `i2c.h`.

#### `i2c` **type**

Represents an I2C master. It is possible for multiple handles to reference the same master device.

#### `i2c i2c_open(int sda, int scl)`

Open an I2C device attached to an I/O switch configured to use the specified pins. Calling this function will disconnect any previously assigned pins and return them to a high-impedance state.

#### `i2c i2c_open_device(unsigned int device)`

Open an I2C master by base address or ID

#### `void i2c_read(i2c dev_id, unsigned int slave_address, unsigned char* buffer, unsigned int length)`

Issue a read command to the specified slave. `buffer` is an array allocated by the caller of at least length `length`.

#### `void i2c_write(i2c dev_id, unsigned int slave_address, unsigned char* buffer, unsigned int length)`

Issue a write command to the specified slave.

#### `void i2c_close(i2c dev_id)`

Close the I2C device.

### SPI Devices

SPI operates on a synchronous transfer of data so rather than read and write, only a `transfer` function is provided. These functions are all provided by `spi.h`.

#### `spi` **type**

Handle to a SPI master.

#### `spi spi_open(unsigned int spiclk, unsigned int miso, unsigned int mosi, unsigned int ss)`

Opens a SPI master on the specified pins. If a pin is not needed for a device, `-1` can be passed in to leave it unconnected.

---

`spi spi_open_device(unsigned int device)`

Opens a SPI master by base address or device ID.

`spi spi_configure(spi dev_id, unsigned int clk_phase, unsigned int clk_polarity)`

Configures the SPI master with the specified clock phase and polarity. These settings are global to all handles to a SPI master.

`void spi_transfer(spi dev_id, const char* write_data, char* read_data, unsigned int length);`

Transfer bytes to and from the SPI slave. Both `write_data` and `write_data` should be allocated by the caller and NULL. Buffers should be at least of length `length`.

`void spi_close(spi dev_id)`

Closes a SPI master

## Timer Devices

Timer devices serve two purposes. They can either be used to output PWM signals or as program timers for inserting accurate delays. It is not possible to use these functions simultaneously and attempting to `delay` while PWM is in operation will result in undefined behavior. All of these functions are in `timer.h`.

`timer` **type**

Handle to an AXI timer

`timer timer_open(unsigned int pin)`

Open an AXI timer attached to the specified pin

`timer timer_open_device(unsigned int device)`

Open an AXI timer by address or device ID

`void timer_delay(timer dev_id, unsigned int cycles)`

Delay the program by a specified number of cycles

`void timer_pwm_generate(timer dev_id, unsigned int period, unsigned int pulse)`

Generate a PWM signal using the specified timer

---

**`void timer_pwm_stop(timer dev_id)`**

Stop the PWM output

**`void timer_close(timer dev_id)`**

Close the specified timer

**`void delay_us(unsigned int us)`**

Delay the program by a number of microseconds using the default delay timer (timer index 0).

**`void delay_ms(unsigned int ms)`**

Delay the program by a number of milliseconds using the default delay timer (timer index 0).

## UART Devices

This device driver controls a UART master.

**`uart type`**

Handle to a UART master device.

**`uart uart_open(unsigned int tx, unsigned int rx)`**

Open a UART device on the specified pins

**`uart uart_open_device(unsigned int device)`**

Open a UART device by base address or index

**`void uart_read(uart dev_id, char* read_data, unsigned int length)`**

Read a fixed length of data from the UART

**`void uart_write(uart dev_id, char* write_data, unsigned int length)`**

Write a block of data to the UART.

**`void uart_close(uart dev_id)`**

Close the handle.

### 2.5.4 PS/PL interface

#### Allocate

The `pynq.allocate` function is used to allocate memory that will be used by IP in the programmable logic.

IP connected to the AXI Master (HP or ACP ports) has access to PS DRAM. Before IP in the PL accesses DRAM, some memory must first be allocated (reserved) for the IP to use and the size, and address of the memory passed to the IP. An array in Python, or Numpy, will be allocated somewhere in virtual memory. The physical memory address of the allocated memory must be provided to IP in the PL.

`pynq.allocate` allocates memory which is physically contiguous and returns a `pynq.Buffer` object representing the allocated buffer. The buffer is a numpy array for use with other Python libraries and also provides a `.device_address` property which contains the physical address for use with IP. For backwards compatibility a `.physical_address` property is also provided

The allocate function uses the same signature as `numpy.ndarray` allowing for any shape and data-type supported by numpy to be used with PYNQ.

#### Buffer

The `Buffer` object returned is a sub-class of `numpy.ndarray` with additional properties and methods suited for use with the programmable logic.

- `device_address` is the address that should be passed to the programmable logic to access the buffer
- `coherent` is True if the buffer is cache-coherent between the PS and PL
- `flush` flushes a non-coherent buffer ensuring that any changes by the PS are visible to the PL
- `invalidate` invalidates a non-coherent buffer ensuring any changes by the PL are visible to the PS

#### Example

Create a contiguous array of 5 32-bit unsigned integers

```python
from pynq import allocate
input_buffer = allocate(shape=(5,), dtype='u4')
```

`device_address` property of the buffer

```python
input_buffer.device_address
```

Writing data to the buffer:

```python
input_buffer[:] = range(5)
```

Flushing the buffer to ensure the updated data is visible to the programmable logic:

```python
input_buffer.flush()
```

More information about memory allocation can be found in the *pynq.buffer Module* section in the library reference.

### Interrupt

The `Interrupt` class represents a single interrupt pin in the block design. It mimics a python `Event` by having a single `wait` function that blocks until the interrupt is raised. The event will be cleared automatically when the interrupt is cleared. To construct an event, pass in fully qualified path to the pin in the block diagram, e.g. `'my_ip/interrupt'` as the only argument.

An interrupt is only enabled for as long there is a thread or coroutine waiting on the corresponding event. The recommended approach to using interrupts is to wait in a loop, checking and clearing the interrupt registers in the IP before resuming the wait. As an example, the AxiGPIO class uses this approach to wait for a desired value to be present.

```python
class AxiGPIO(DefaultIP):
    # Rest of class definition

    def wait_for_level(self, value):
        while self.read() != value:
            self._interrupt.wait()
            # Clear interrupt
            self._mmio.write(IP_ISR, 0x1)
```
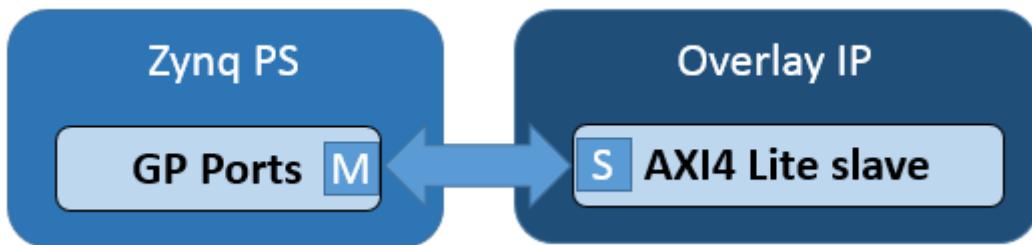
### MMIO

The *MMIO* class allows a Python object to access addresses in the system memory mapped. In particular, registers and address space of peripherals in the PL can be accessed.

### AXI GP ports

In an overlay, peripherals connected to the AXI General Purpose ports will have their registers or address space mapped into the system memory map. With PYNQ, the register, or address space of an IP can be accessed from Python using the *MMIO* class.



MMIO provides a simple but powerful way to access and control peripherals. For simple peripherals with a small number of memory accesses, or where performance is not critical, MMIO is usually sufficient for most developers. If performance is critical, or large amounts of data need to be transferred between PS and PL, using the Zynq HP interfaces with DMA IP and the PYNQ DMA class may be more appropriate.

### Example

In this example, data is written to an IP and read back from the same address.

```
IP_BASE_ADDRESS = 0x40000000
ADDRESS_RANGE = 0x1000
ADDRESS_OFFSET = 0x10

from pynq import MMIO
mmio = MMIO(IP_BASE_ADDRESS, ADDRESS_RANGE)

data = 0xdeadbeef
mmio.write(ADDRESS_OFFSET, data)
result = mmio.read(ADDRESS_OFFSET)
```

This example assumes the memory mapped area defined for the MMIO, from `0x40000000` to `0x40001000`, is accessible to the PS.
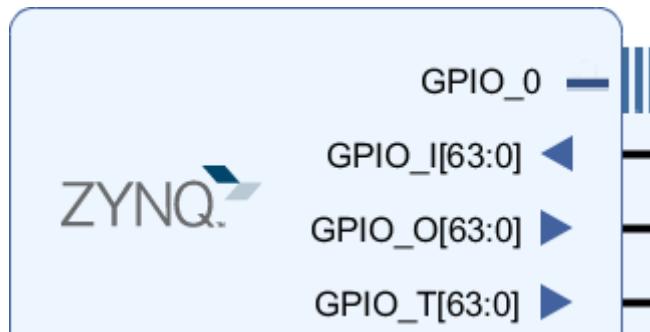
More information about the MMIO module can be found in the *pynq.mmio Module* sections

### PS GPIO

The Zynq device has up to 64 GPIO from PS to PL. These can be used for simple control type operations. For example, in the base overlay, the PS GPIO wires are used as the reset signals for the IOPs. The PS GPIO are a very simple interface and there is no IP required in the PL to use them.

The *GPIO* class is used to control the PS GPIO. Note that *AXI* GPIO are controlled by the *AxiGPIO* class.

### Block Diagram



### Linux GPIO

The PS GPIO use a Linux kernel module to control the GPIO. This means that the operating systems assign a number to the GPIO at run time. Before using the PS GPIO, the Linux pin number must be mapped to the Python GPIO instance.

The `get_gpio_pin()` function which is part of the GPIO class is used to map the PS pin number to the Linux pin number. See the example below on how it can be used.

### Example

The PS GPIO need to be connected to something in an overlay before they can be used. The example below is for illustration. It shows a code snippet that needs an appropriate overlay with the PS GPIO connected to something.

```python
from pynq import GPIO

output = GPIO(GPIO.get_gpio_pin(0), 'out')
input = GPIO(GPIO.get_gpio_pin(1), 'in')

output.write(0)
input.read()
```

More information about the GPIO module and the API for reading, writing and waiting for interrupts can be found in the *pynq.gpio Module* sections

## 2.5.5 PS control

### PMBus

PYNQ provides access to voltage and current sensors provided on many boards using PMBus or other protocols supported by the Linux kernel. PYNQ uses the libsensors API (https://github.com/lm-sensors/lm-sensors) to provide access to monitoring sensors.

### `pynq.pmbus` API

All sensors can be found using the `pynq.get_rails()` function which returns a dictionary mapping the name of the voltage rail to a `Rail` class. Each `Rail` has members for the `voltage`, `current` and `power` sensors, the current reading of which can be obtained from the `value` attribute.

### The `DataRecorder`

The other aspect of the PMBus library is the `DataRecorder` class which provides a simple way to record the values of one or more sensors during a test. A `DataRecorder` is constructed with the sensors to be monitored and will ultimately produce a pandas `DataFrame` as the result. The `record(sample_interval)` function begins the recording with the sample rate specified as the interval in seconds. The `stop()` function ends the recording. If the `record` function is used in a `with` block the `stop` will be called automatically at the end of the block ensuring that the monitoring thread is always terminated even in the presence of exceptions. Each sample in the result is indexed by a timestamp and contains a session identifier in addition to the values. This identifier starts at `0` and is incremented each time that `record` is called on a recorder or when the `mark()` function is called. This identifier is designed to allow different parts or runs of a test to be differentiated in further analysis.

### Example

```python
from pynq import get_rails, DataRecorder

rails = get_rails()
recorder = DataRecorder(rails['12V'].power)

with recorder.record(0.2): # Sample every 200 ms
    # Perform the first part of the test
    recorder.mark()
    # Perform the second part of the test

results = recorder.frame
```

### Board Support

For full support on a board a custom configuration file is required for libsensors to identify which voltage rails are attached to which sensors which should be copied to `/etc/sensors.d`. The PYNQ repository contains a configuration for the ZCU104 board. For details on the format of this file see both the ZCU104 configuration in `boards/ZCU104/packages/sensorconf` directory and the lm-sensors documentation at the link in the introduction.

## 2.5.6 PL control

### Overlay

The *Overlay* class is used to load PYNQ overlays to the PL, and manage and control existing overlays. The class is instantiated with the .bit file for an overlay. By default the overlay Tcl file will be parsed, and the bitstream will be downloaded to the PL. This means that to use the overlay class, a .bit and .tcl must be provided for an overlay.

To instantiate the Overlay only without downloading the .bit file, pass the parameter *download=False* when instantiating the Overlay class.

On downloading the bitstream, the clock settings provided in the overlay .tcl file will also be applied before the bitstream is downloaded.

### Examples

```python
from pynq import Overlay

base = Overlay("base.bit") # bitstream implicitly downloaded to PL
```

The .bit file path can be provided as a relative, or absolute path. The Overlay class will also search the packages directory for installed packages, and download an overlay found in this location. The .bit file is used to locate the package.

```python
base = Overlay("base.bit", download=False) # Overlay is instantiated, but bitstream
→is not downloaded to PL

base.download() # Explicitly download bitstream to PL

base.is_loaded() # Checks if a bitstream is loaded

base.reset() # Resets all the dictionaries kept int he overlay

base.load_ip_data(myIP, data) # Provides a function to write data to the memory space
→of an IP
                              # data is assumed to be in binary format
```

The *ip_dict* contains a list of IP in the overlay, and can be used to determine the IP driver, physical address, version, if GPIO, or interrupts are connected to the IP.

```
base.ip_dict
```

More information about the Overlay module can be found in the *pynq.overlay Module* sections.

### PL and Bitstream classes

The *PL* is mainly an internal class used by the Overlay class. The PL class launches a *PL server* which manages the loaded overlay. The PL server can stop multiple overlays from different applications from overwriting the currently loaded overlay.

The overlay Tcl file is parsed by the PL class to generate the IP, clock, interrupts, and gpio dictionaries (lists of information about IP, clocks and signals in the overlay).

The *Bitstream* class can be found in the pl.py source file, and can be used instead of the overlay class to download a bitstream file to the PL without requiring an overlay Tcl file. This can be used for testing, but these attributes can also be accessed through the Overlay class (which inherits from this class). Using the Overlay class is the recommended way to access these attributes.

### Examples

### PL

```
PL.timestamp # Get the timestamp when the current overlay was loaded

PL.ip_dict # List IP in the overlay

PL.gpio_dict # List GPIO in the overlay

PL.interrupt_controllers # List interrupt controllers in the overlay

PL.interrupt_pins # List interrupt pins in the overlay

PL.hierarchy_dict # List the hierarchies in the overlay
```

### Bitstream

```python
from pynq import Bitstream

bit = Bitstream("base.bit") # No overlay Tcl file required

bit.download()

bit.bitfile_name
```

'/opt/python3.6/lib/python3.6/site-packages/pynq/overlays/base/base.bit'

More information about the PL module can be found in the *pynq.pl Module* sections.

## 2.6 Overlay Design Methodology

As described in the PYNQ introduction, overlays are analogous to software libraries. A programmer can download overlays into the Zynq® PL at runtime to provide functionality required by the software application.

An *overlay* is a class of Programmable Logic design. Programmable Logic designs are usually highly optimized for a specific task. Overlays however, are designed to be configurable, and reusable for broad set of applications. A PYNQ overlay will have a Python interface, allowing a software programmer to use it like any other Python package.

A software programmer can use an overlay, but will not usually create overlay, as this usually requires a high degree of hardware design expertise.

There are a number of components required in the process of creating an overlay:

- Board Settings
- PS-PL Interface
- MicroBlaze Soft Processors
- Python/C Integration
- Python AsyncIO
- Python Overlay API
- Python Packaging

This section will give an overview of the process of creating an overlay and integrating it into PYNQ, but will not cover the hardware design process in detail. Hardware design will be familiar to Zynq, and FPGA hardware developers.

### 2.6.1 Overlay Design

An overlay consists of two main parts; the PL design (bitstream) and the project block diagram Tcl file. Overlay design is a specialized task for hardware engineers. This section assumes the reader has some experience with digital design, building Zynq systems, and with the Vivado design tools.

#### PL Design

The Xilinx® Vivado software is used to create a Zynq design. A *bitstream* or *binary* file (.bit file) will be generated that can be used to program the Zynq PL.

A free WebPack version of Vivado is available to build overlays. https://www.xilinx.com/products/design-tools/vivado/vivado-webpack.html

The hardware designer is encouraged to support programmability in the IP used in a PYNQ overlays. Once the IP has been created, the PL design is carried out in the same way as any other Zynq design. IP in an overlay that can be controlled by PYNQ will be memory mapped, connected to GPIO. IP may also have a master connection to the PL. PYNQ provides Python libraries to interface to the PL design and which can be used to create their own drivers. The Python API for an overlay will be and will be covered in the next sections.

#### Overlay Tcl file

The Tcl from the Vivado IP Integrator block design for the PL design is used by PYNQ to automatically identify the Zynq system configuration, IP including versions, interrupts, resets, and other control signals. Based on this information, some parts of the system configuration can be automatically modified from PYNQ, drivers can be automatically assigned, features can be enabled or disabled, and signals can be connected to corresponding Python methods.

The Tcl file must be generated and provided with the bitstream file as part of an overlay. The Tcl file can be generated in Vivado by *exporting* the IP Integrator block diagram at the end of the overlay design process. The Tcl file should be provided with a bitstream when downloading an overlay. The PYNQ PL class will automatically parse the Tcl.

A custom, or manually created Tcl file can be used to build a Vivado project, but Vivado should be used to generate and export the Tcl file for the block diagram. This automatically generated Tcl should ensure that it can be parsed correctly by the PYNQ.

To generate the Tcl for the Block Diagram from the Vivado GUI:

- Click **File > Export > Block Design**

Or, run the following in the Tcl console:

```
write_bd_tcl
```

The Tcl filename should match the .bit filename. For example, *my_overlay.bit* and *my_overlay.tcl*.

The Tcl is parsed when the overlay is instantiated and downloaded.

```python
from pynq import Overlay
ol = Overlay("base.bit") # Tcl is parsed here
```

An error will be displayed if a Tcl is not available when attempting to download an overlay, or if the Tcl filename does not match the .bit file name.

### Programmability

An overlay should have post-bitstream programmability to allow customization of the system. A number of reusable PYNQ IP blocks are available to support programmability. For example, a PYNQ MicroBlaze can be used on Pmod, and Arduino interfaces. IP from the various overlays can be reused to provide run-time configurability.

### Zynq PS Settings

A Vivado project for a Zynq design consists of two parts; the PL design, and the PS configuration settings.

The PYNQ image which is used to boot the board configures the Zynq PS at boot time. This will fix most of the PS configuration, including setup of DRAM, and enabling of the Zynq PS peripherals, including SD card, Ethernet, USB and UART which are used by PYNQ.

The PS configuration also includes settings for system clocks, including the clocks used in the PL. The PL clocks can be programmed at runtime to match the requirements of the overlay. This is managed automatically by the PYNQ Overlay class.

During the process of downloading a new overlay, the clock configuration will be parsed from the overlay's Tcl file. The new clock settings for the overlay will be applied automatically before the overlay is downloaded.

### Existing Overlays

Existing overlays can be used as a starting point to create a new overlay. The *base* overlay can be found in the *boards* directory in the PYNQ repository, and includes reference IP for peripherals on the board:

```
<PYNQ repository>/boards/<board>/base
```

A makefile exists in each folder that can be used to rebuild the Vivado project and generate the bitstream and Tcl for the overlay. (On windows, instead of using *make*, the Tcl file can be sourced from Vivado.)

The bitstream and Tcl for the overlay are available on the board, and also in the GitHub project repository:

```
<PYNQ repository>/boards/<board>/base
```

## 2.6.2 Board Settings

**Base overlay project**

The source files for the *base* overlay for supported boards can be found in the PYNQ GitHub. The project can be rebuilt using the makefile/TCL available here:

```
<GitHub repository>/boards/<board>/base
```

The base design can be used as a starting point to create a new design.

**Vivado board files**

Vivado board files contain the configuration for a board that is required when creating a new project in Vivado. For most of the Xilinx boards (for example, ZCU104), the board files have already been included in Vivado; users can simply choose the corresponding board when they create a new project. For some other boards (for example, Pynq-Z1 and Pynq-Z2), the corresponding board files can be downloaded as shown below.

- Download the Pynq-Z1 board files
- Download the Pynq-Z2 board files

Installing these files in Vivado allows the board to be selected when creating a new project. This will configure the Zynq PS settings.

To install the board files, extract, and copy the board files folder to:

```
<Xilinx installation directory>\Vivado\<version>\data\boards
```

If Vivado is open, it must be restart to load in the new project files before a new project can be created.
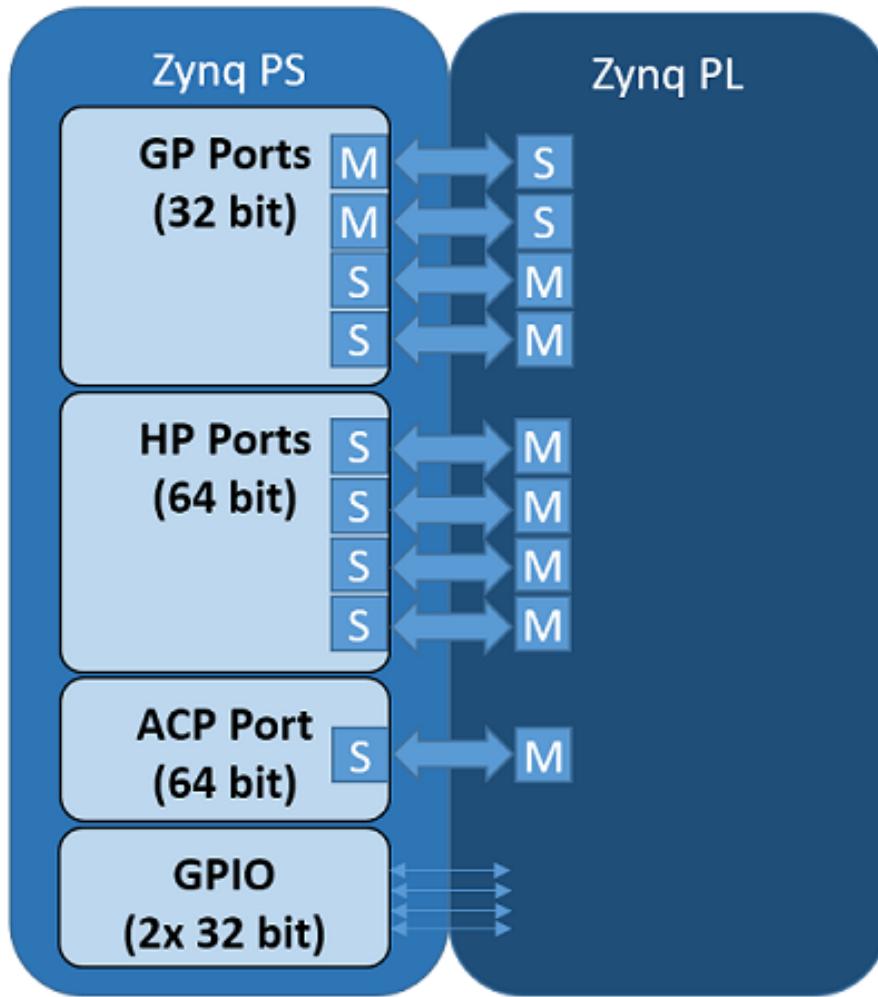
**XDC constraints file**

Please see below for a list of constraint files.

- Download the Pynq-Z1 Master XDC constraints
- Download the Pynq-Z2 Master XDC constraints

## 2.6.3 PS/PL Interfaces

The Zynq has 9 AXI interfaces between the PS and the PL. On the PL side, there are 4x AXI Master HP (High Performance) ports, 2x AXI GP (General Purpose) ports, 2x AXI Slave GP ports and 1x AXI Master ACP port. There are also GPIO controllers in the PS that are connected to the PL.

There are four `pynq` classes that are used to manage data movement between the Zynq PS (including the PS DRAM) and PL interfaces.

- GPIO - General Purpose Input/Output

- MMIO - Memory Mapped IO

- allocate - Memory allocation

- DMA - Direct Memory Access

The class used depends on the Zynq PS interface the IP is connected to, and the interface of the IP.

Python code running on PYNQ can access IP connected to an AXI Slave connected to a GP port. *MMIO* can be used to do this.

IP connected to an AXI Master port is not under direct control of the PS. The AXI Master port allows the IP to access DRAM directly. Before doing this, memory should be allocated for the IP to use. The *allocate* function can be used to do this. For higher performance data transfer between PS DRAM and an IP, DMAs can be used. PYNQ provides a DMA class.

When designing your own overlay, you need to consider the type of IP you need, and how it will connect to the PS. You should then be able to determine which classes you need to use the IP.

### PS GPIO

There are 64 GPIO (wires) from the Zynq PS to PL.

PS GPIO wires from the PS can be used as a very simple way to communicate between PS and PL. For example, GPIO can be used as control signals for resets, or interrupts.

IP does not have to be mapped into the system memory map to be connected to GPIO.

More information about using PS GPIO can be found in the *PS GPIO* section.

### MMIO

Any IP connected to the AXI Slave GP port will be mapped into the system memory map. MMIO can be used read/write a memory mapped location. A MMIO read or write command is a single transaction to transfer 32 bits of data to or from a memory location. As burst instructions are not supported, MMIO is most appropriate for reading and writing small amounts of data to/from IP connect to the AXI Slave GP ports.

More information about using MMIO can be found in the *MMIO* section.

### allocate

Memory must be allocated before it can be accessed by the IP. `allocate` allows memory buffers to be allocated. The `allocate` function allocates a contiguous memory buffer which allows efficient transfers of data between PS and PL. Python or other code running in Linux on the PS can access the memory buffer directly.

As PYNQ is running Linux, the buffer will exist in the Linux virtual memory. The Zynq AXI Slave ports allow an AXI-master IP in an overlay to access physical memory. The numpy array returned can also provide the physical memory pointer to the buffer which can be sent to an IP in the overlay. The physical address is stored in the `device_address` property of the allocated memory buffer instance. An IP in an overlay can then access the same buffer using the physical address.

More information about using allocate can be found in the *Allocate* section.

### DMA

AXI stream interfaces are commonly used for high performance streaming applications. AXI streams can be used with Zynq AXI HP ports via a DMA.

The `pynq` DMA class supports the AXI Direct Memory Access IP. This allows data to be read from DRAM, and sent to an AXI stream, or received from a stream and written to DRAM.

More information about using DMA can be found in the *DMA* section.

### Interrupt

There are dedicated interrupts which are linked with asyncio events in the python environment. To integrate into the PYNQ framework Dedicated interrupts must be attached to an AXI Interrupt controller which is in turn attached to the first interrupt line to the processing system. If more than 32 interrupts are required then AXI interrupt controllers can be cascaded. This arrangement leaves the other interrupts free for IP not controlled by PYNQ directly such as SDSoC accelerators.

Interrupts are managed by the Interrupt class, and the implementation is built on top of *asyncio*, part of the Python standard library.

More information about using the Interrupt class can be found in the *Interrupt* section.

For more details on *asyncio*, how it can be used with PYNQ see the *PYNQ and Asyncio* section.

### 2.6.4 PYNQ MicroBlaze Subsystem

The PYNQ MicroBlaze subsystem gives flexibility to support a wide range of hardware peripherals from Python. The PYNQ MicroBlaze is intended as an offload processor, and can deal with the low level communication protocols and data processing and provides data from a sensor that can be accessed from Python. The subsystem is deterministic, and is suitable for real-time control.

MicroBlaze applications will typically be developed in C or C++, and will run bare-metal.

The following sections show how to develop applications for the MicroBlaze soft processors running inside an overlay.

#### Memory Architecture

Each PYNQ MicroBlaze has local memory (implemented in Xilinx BRAMs) and a connection to the PS DDR memory.

The PYNQ MicroBlaze instruction and data memory is implemented in a dual port Block RAM, with one port connected to the PYNQ MicroBlaze, and the other to the ARM processor. This allows an executable binary file to be written from the ARM to the PYNQ MicroBlaze instruction memory. The PYNQ MicroBlaze can also be reset by the ARM, allowing the PYNQ MicroBlaze to start executing the new program.

The PYNQ MicroBlaze data memory, either in local memory, or in DDR memory, can be used as a mailbox for communication and data exchanges between the ARM processor and the PYNQ MicroBlaze.

DDR memory is managed by the Linux kernel running on the Cortex-A9s. Therefore, the PYNQ MicroBlaze must first be allocated memory regions to access DRAM. The PYNQ `allocate` function is used to allocate memory in Linux. It also provides the physical address of the memory. A PYNQ applications can send the physical address to a PYNQ MicroBlaze, and the PYNQ MicroBlaze can then access the buffer.

#### DDR Memory

The PYNQ MicroBlazes are connected to the DDR memory via the General Purpose AXI slave port. This is a direct connection, so it is only suitable for simple data transfers from the PYNQ MicroBlaze. The MicroBlaze can attempt to read or write the DDR as quickly as possible in a loop, but there is no support for bursts, or streaming data.

#### PYNQ MicroBlaze Memory Map

The local PYNQ MicroBlaze memory is 64KB of shared data and instruction memory. Instruction memory for the PYNQ MicroBlaze starts at address 0x0.

PYNQ and the application running on the PYNQ MicroBlaze can write to anywhere in the shared memory space. You should be careful not to write to the instruction memory unintentionally as this will corrupt the running application.

When building the MicroBlaze project, the compiler will only ensure that *allocated* stack and heap of the application fit into the BRAM and DDR if used. For communication between the ARM and the MicroBlaze, a part of the shared memory space must also be reserved within the MicroBlaze address space.

There is no memory management in the PYNQ MicroBlaze. You must ensure the application, including stack and heap, do not overflow into the defined data area. Remember that declaring a stack and heap size only allocates space to the stack and heap. No boundary is created, so if sufficient space was not allocated, the stack and heap may overflow and corrupt your application.

If you need to modify the stack and heap for an application, the linker script can be found in the `<project_directory>/src/` directory.

It is recommended to follow the same convention for data communication between the two processors via a MAIL-BOX.

| | |
|---|---|
| Instruction and data memory start | 0x0 |
| Instruction and data memory size | 0xf000 |
| Shared mailbox memory start | 0xf000 |
| Shared mailbox memory size | 0x1000 |
| Shared mailbox Command Address | 0xfffc |

These MAILBOX values for a PYNQ MicroBlaze application are defined here:

```
<PYNQ repository>/boards/sw_repo/pynqmb/src/circular_buffer.h
```

The corresponding Python constants are defined here:

```
<PYNQ repository>/pynq/lib/pmod/constants.py
<PYNQ repository>/pynq/lib/arduino/constants.py
<PYNQ repository>/pynq/lib/rpi/constants.py
```

The following example explains how Python could initiate a read from a peripheral connected to a PYNQ MicroBlaze.

1. Python writes a read command (e.g. 0x3) to the mailbox command address (0xfffc).

2. MicroBlaze application checks the command address, and reads and decodes the command.

3. MicroBlaze performs a read from the peripheral and places the data at the mailbox base address (0xf000).

4. MicroBlaze writes 0x0 to the mailbox command address (0xfffc) to confirm transaction is complete.

5. Python checks the command address (0xfffc), and sees that the MicroBlaze has written 0x0, indicating the read is complete, and data is available.

6. Python reads the data in the mailbox base address (0xf000), completing the read.

### Running Code on Different MicroBlazes

The MicroBlaze local BRAM memory is mapped into the MicroBlaze address space, and also to the ARM address space. These address spaces are independent, so the local memory will be located at different addresses in each memory space. Some example mappings are shown below to highlight the address translation between MicroBlaze and ARM's memory spaces.

| MicroBlaze Base Address | MicroBlaze Address Space | ARM Equivalent Address Space |
|---|---|---|
| 0x4000_0000 | 0x0000_0000 - 0x0000_ffff | 0x4000_0000 - 0x4000_ffff |
| 0x4200_0000 | 0x0000_0000 - 0x0000_ffff | 0x4200_0000 - 0x4200_ffff |
| 0x4400_0000 | 0x0000_0000 - 0x0000_ffff | 0x4400_0000 - 0x4400_ffff |

Note that each MicroBlaze has the same range for its address space. However, the location of the address space in the ARM memory map is different for each PYNQ MicroBlaze. As the MicroBlaze address space is the same for each PYNQ MicroBlaze, any binary compiled for one PYNQ MicroBlaze will work on another PYNQ MicroBlaze.

For example, suppose a PYNQ MicroBlaze exists at 0x4000_0000, and a second PYNQ MicroBlaze exists at 0x4200_0000. The same binary can run on the first PYNQ MicroBlaze by writing the binary from python to the address space `0x4000_0000`, and on the second PYNQ MicroBlaze by writing to `0x4200_0000`.

### Building Applications

There are a number of steps required before you can start writing your own software for a PYNQ MicroBlaze. This document will describe the PYNQ MicroBlaze architecture, and how to set up and build the required software projects to allow you to write your own application for the MicroBlaze inside an PYNQ MicroBlaze.

Xilinx® SDK projects can be created manually using the SDK GUI, or software can be built using a Makefile flow. Starting from image v2.1, users can also directly use the Jupyter notebook to program the PYNQ MicroBlaze; more examples can be found in

```
<PYNQ dashboard>/base/microblaze
```

### MicroBlaze Processors

As described in the previous section, a PYNQ MicroBlaze can be used as a flexible controller for different types of external peripherals. The ARM® Cortex®-A9 is an application processor, which runs PYNQ and Jupyter notebook on a Linux OS. This scenario is not well suited to real-time applications, which is a common requirement for an embedded systems. In the base overlay there are three PYNQ MicroBlazes. As well as acting as a flexible controller, a PYNQ MicroBlaze can be used as dedicated real-time controller.

PYNQ MicroBlazes can also be used standalone to offload some processing from the main processor. However, note that the MicroBlaze processor inside a PYNQ MicroBlaze in the base overlay is running at 100 MHz, compared to the Dual-Core ARM Cortex-A9 running at 650 MHz. The clock speed, and different processor architectures and features should be taken into account when offloading pure application code. e.g. Vector processing on the ARM Cortex-A9 Neon processing unit will be much more efficient than running on the MicroBlaze. The MicroBlaze is most appropriate for low-level, background, or real-time applications.

### Software Requirements

Xilinx SDK (Software Development Kit) contains the MicroBlaze cross-compiler which can be used to build software for the MicroBlaze inside a PYNQ MicroBlaze. SDK is available for free as part of the Xilinx Vivado WebPack.

The full source code for all supported PYNQ MicroBlaze peripherals is available from the project GitHub. PYNQ ships with precompiled PYNQ MicroBlaze executables to support various peripherals (see *PYNQ Libraries*), so Xilinx software is only needed if you intend to modify existing code, or build your own PYNQ MicroBlaze applications/peripheral drivers.

PYNQ releases are built using:

| Release version | Vivado and SDK |
|---|---|
| v1.4 | 2015.4 |
| v2.0 | 2016.1 |
| v2.1 | 2017.4 |

It is recommended to use the same version to rebuild existing Vivado and SDK projects. If you only intend to build software, you will only need to install SDK. The full Vivado and SDK installation is only required to modify or design new overlays. You can use the Vivado HLx Web Install Client and select SDK and/or Vivado during the installation.

### Compiling Projects

Software executables run on the MicroBlaze inside a PYNQ MicroBlaze. Code for the MicroBlaze can be written in C or C++ and compiled using Xilinx SDK .

You can pull or clone the PYNQ repository, and all the driver source and project files can be found in `<PYNQ repository>\pynq\lib\<driver_group_name>\<project_directory>`.

### SDK Application, Board Support Package, Hardware Platform

Each SDK application project requires a BSP project (Board Support Package), and a hardware platform project. The application project will include the user code (C/C++). The Application project is linked to a BSP. The BSP (Board Support Package) contains software libraries and drivers to support the underlying peripherals in the system.

Internally, the BSP is linked to a Hardware Platform. A Hardware Platform defines the peripherals in the PYNQ MicroBlaze subsystem, and the memory map of the system. It is used by the BSP to build software libraries to support the underlying hardware.

All *Application* projects can be compiled from the command line using makefiles, or imported into the SDK GUI.

You can also use existing projects as a starting point to create your own project.

### Board Support Package

A Board Support Package (BSP) includes software libraries for peripherals in the system. For example, the SDK projects for Pmod and Arduino peripherals require the following 2 BSPs:

BSP for the Arduino PYNQ MicroBlaze:

> `<PYNQ repository>/pynq/lib/arduino/bsp_iop_arduino/`

BSP for the Pmod PYNQ MicroBlaze:

> `<PYNQ repository>/pynq/lib/pmod/bsp_iop_pmod`

A BSP is specific to a processor subsystem. There can be many BSPs associated with an overlay, depending on the types of processors available in the system.

An application for the Pmod PYNQ MicroBlaze will be linked to the Pmod PYNQ MicroBlaze BSP. As the two Pmod PYNQ MicroBlazes are identical, an application written for one Pmod PYNQ MicroBlaze can run on the other Pmod PYNQ MicroBlaze.

An Arduino application will be linked to the Arduino PYNQ MicroBlaze BSP.

### Building the Projects

To build all the software projects, for example, you can run the corresponding makefile:

> `<PYNQ repository>/pynq/lib/arduino/makefile`
>
> `<PYNQ repository>/pynq/lib/pmod/makefile`

Application projects for peripherals that ship with PYNQ (e.g. Pmod and Arduino peripherals) can also be found in the same location. Each project is contained in a separate folder.

The makefile compiles the application projects based on the BSP provided in the correct location.

The makefile requires SDK to be installed, and can be run from Windows, or Linux.

To run `make` from Windows, open SDK, and choose a temporary workspace (make sure this path is external to the downloaded PYNQ repository). From the *Xilinx Tools* menu, select *Launch Shell*.

In Linux, open a terminal, and source the SDK tools.

From either the Windows Shell, or the Linux terminal, navigate to the sdk folder in your local copy of the PYNQ repository:

The following example shows how to run `make` in `<PYNQ repository>/pynq/lib/pmod/`:



This will clean all the existing compiled binaries (bin files), and rebuild all the application projects.

```
Building target: pmod_grove_haptic_motor.elf
Invoking: MicroBlaze gcc linker
mb-gcc -Wl,-T -Wl,../../src/lscript.ld -L../../bsp_pmod/iop1_mb/lib -mlittle-endian -mcpu=v9.5 -mxl-soft-mul -Wl,--no-relax
-o "pmod_grove_haptic_motor.elf"  ./src/pmod_grove_haptic_motor.o   -Wl,--start-group,-lxil,-lgcc,-lc,--end-group
Finished building target: pmod_grove_haptic_motor.elf

Invoking: MicroBlaze Print Size
mb-size pmod_grove_haptic_motor.elf  |tee "pmod_grove_haptic_motor.elf.size"
   text    data     bss     dec     hex filename
  14196     360    3316   17872    45d0 pmod_grove_haptic_motor.elf
Finished building: pmod_grove_haptic_motor.elf.size

Invoking: MicroBlaze Bin Gen
mb-objcopy -O binary pmod_grove_haptic_motor.elf pmod_grove_haptic_motor.bin
Finished building: pmod_grove_haptic_motor.bin

make[1]: Leaving directory `/home/yunq/Pynq-Python/PYNQ-rock/pynq/lib/pmod/pmod_grove_haptic_motor/Debug'
cd pmod_grove_th02/Debug && make clean && make
make[1]: Entering directory `/home/yunq/Pynq-Python/PYNQ-rock/pynq/lib/pmod/pmod_grove_th02/Debug'
rm -rf  ./src/pmod_grove_th02.o  ./src/pmod_grove_th02.d  pmod_grove_th02.elf.size  pmod_grove_th02.elf pmod_grove_th02.
in

make[1]: Leaving directory `/home/yunq/Pynq-Python/PYNQ-rock/pynq/lib/pmod/pmod_grove_th02/Debug'
make[1]: Entering directory `/home/yunq/Pynq-Python/PYNQ-rock/pynq/lib/pmod/pmod_grove_th02/Debug'
Building file: ../src/pmod_grove_th02.c
Invoking: MicroBlaze gcc compiler
mb-gcc -Wall -O1 -g3 -c -fmessage-length=0 -MT"src/pmod_grove_th02.o" -I../../bsp_pmod/iop1_mb/include -mlittle-endian
cpu=v9.5 -mxl-soft-mul -Wl,--no-relax -MMD -MP -MF"src/pmod_grove_th02.d" -MT"src/pmod_grove_th02.d" -o "src/pmod_grove_
th02.o" "../src/pmod_grove_th02.c"
Finished building: ../src/pmod_grove_th02.c

Building target: pmod_grove_th02.elf
Invoking: MicroBlaze gcc linker
mb-gcc -Wl,-T -Wl,../../src/lscript.ld -L../../bsp_pmod/iop1_mb/lib -mlittle-endian -mcpu=v9.5 -mxl-soft-mul -Wl,--no-relax
-o "pmod_grove_th02.elf"  ./src/pmod_grove_th02.o   -Wl,--start-group,-lxil,-lgcc,-lc,--end-group
Finished building target: pmod_grove_th02.elf

Invoking: MicroBlaze Print Size
mb-size pmod_grove_th02.elf  |tee "pmod_grove_th02.elf.size"
   text    data     bss     dec     hex filename
  14240     360    3312   17912    45f8 pmod_grove_th02.elf
Finished building: pmod_grove_th02.elf.size

Invoking: MicroBlaze Bin Gen
mb-objcopy -O binary pmod_grove_th02.elf pmod_grove_th02.bin
Finished building: pmod_grove_th02.bin

make[1]: Leaving directory `/home/yunq/Pynq-Python/PYNQ-rock/pynq/lib/pmod/pmod_grove_th02/Debug'

Completed Microblaze Projects' Builds

yunq@xsjpsgv106-67% []
```

If you examine the makefile, you can the *BIN_PMOD* variable at the top of the makefile includes all the bin files required by Pmod peripherals. If you want to add your own custom project to the build process, you need to add the project name to the *BIN_PMOD* variable, and save the project in the same location as the other application projects.

Similarly, you have to following the same steps to build Arduino application projects.

In addition, individual projects can be built by navigating to the `<project_directory>/Debug` and running `make`.

**Binary Files**

Compiling code produces an executable file (.elf) along with its binary format (.bin) to be downloaded to a PYNQ MicroBlaze.

A .bin file can be generated from a .elf by running the following command from the SDK shell:

```
mb-objcopy -O binary <input_file>.elf <output_file>.bin
```

This is done automatically by the makefile for the existing application projects. The makefile will also copy all .bin files into the `<PYNQ repository>/pynq/lib/<driver_group_name>/` folder.

**Creating Your Own**

Using the makefile flow, you can use an existing project as a starting point for your own project.

Copy and rename the project, and modify or replace the .c file in the src/ with your C code. The generated .bin file will have the same base name as your C file.
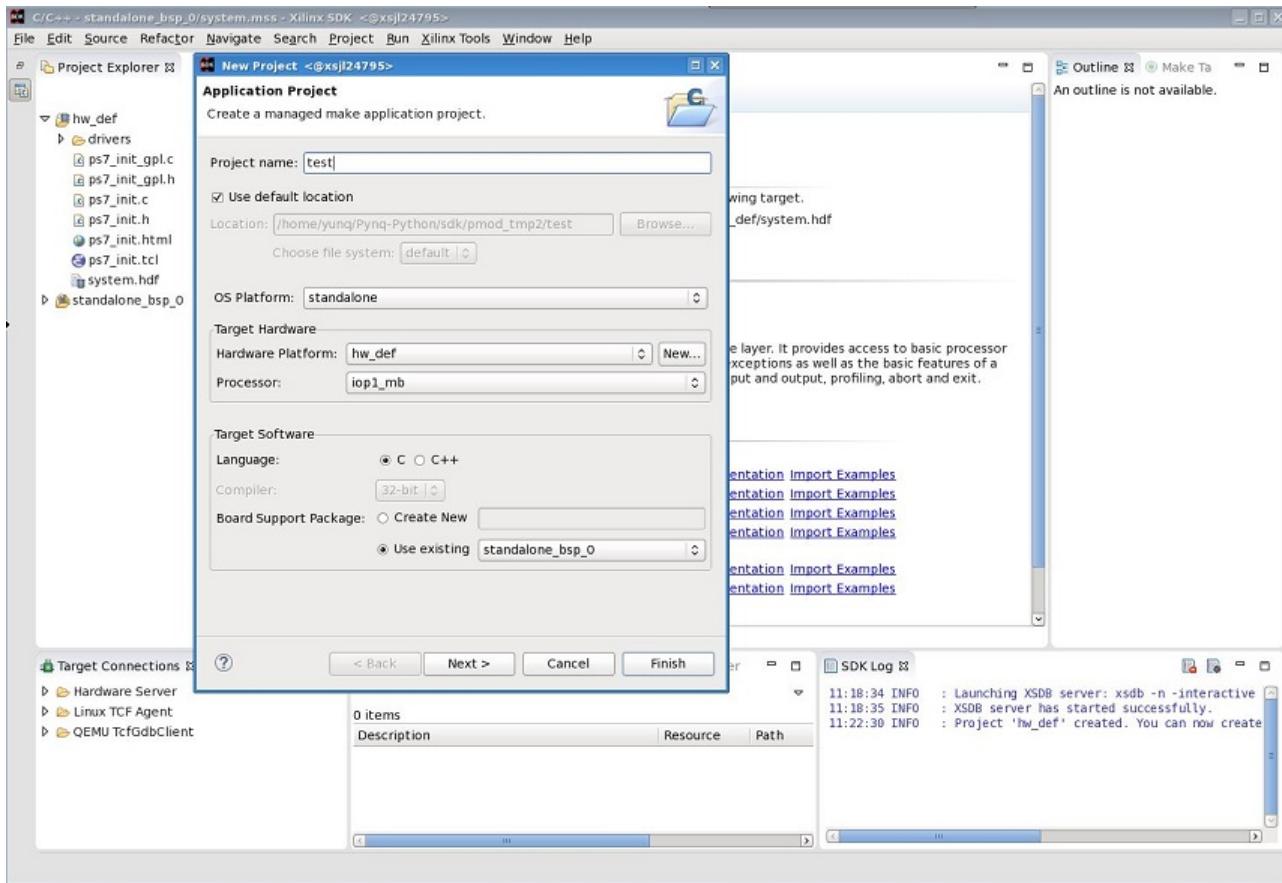
For example, if your C code is `my_peripheral.c`, the generated .elf and .bin will be `my_peripheral.elf` and `my_peripheral.bin`.

The naming convention recommended for peripheral applications is `<pmod|arduino>_<peripheral>`.

You will need to update references from the old project name to your new project name in `<project_directory>/Debug/makefile` and `<project_directory>/Debug/src/subdir.mk`.

If you want your project to build in the main makefile, you should also append the .bin name of your project to the *BIN_PMOD* (or *BIN_ARDUINO*) variable at the top of the makefile.

If you are using the SDK GUI, you can import the Hardware Platform, BSP, and any application projects into your SDK workspace.

The SDK GUI can be used to build and debug your code.

## Writing Applications

The previous section described the software architecture and the software build process. This section will cover how to write the PYNQ MicroBlaze application and also the corresponding Python interface.

The section assumes that the hardware platform and the BSPs have already been generated as detailed in the previous section.

## Header Files and Libraries

A library is provided for the PYNQ MicroBlaze which includes an API for local peripherals (IIC, SPI, Timer, Uart, GPIO), the configurable switch, links to the peripheral addresses, and mappings for the mailbox used in the existing PYNQ MicroBlaze peripheral applications provided with PYNQ. This library can be used to write custom PYNQ MicroBlaze applications.

The PYNQ MicroBlaze can deploy a configurable IO switch. It allows the IO pins to be connected to various types of controllers. The header files associated with the corresponding configurable switch can be found:

```
<PYNQ repository>/boards/ip/io_switch_1.1/drivers/io_switch_v1_0/src
```

The PYNQ MicroBlaze has a dedicated library *pynqmb*. It wraps up low-level functions for ease of use. The header files can be found

```
<PYNQ repository>/boards/sw_repo/pynqmb/src
```

To use these files in a PYNQ MicroBlaze application, include these header file(s) in the C program.

For example:

```
#include "xio_switch.h"
#include "circular_buffer.h"
#include "gpio.h"
```

### Controlling the IO Switch

The IO switch needs to be configured by the PYNQ MicroBlaze application before any peripherals can be used. This can be done statically from within the application, or the application can allow Python to write a switch configuration to shared memory, which can be used to configure the switch.

For Pmod, there are 8 data pins that can be connected to GPIO, SPI, IIC, or Timer. For Arduino, there are 20 shared data pins that can be connected to GPIO, UART, SPI, or Timer. For RPi, there are 28 shared data pins that can be connected to GPIO, UART, SPI, or Timer.

The following function, part of the provided IO switch driver (*xio_switch.h*), can be used to configure the switch from a PYNQ MicroBlaze application.

```
void init_io_switch(void);
void set_pin(int pin_number, u8 pin_type);
```

The function *init_io_switch()* will just set all the pins to GPIO by default. Then users can call *set_pin()* to configure each individual pin. The valid values for the parameter *pin_type* are defined as:

| Pin | Value |
|-----|-------|
| GPIO | 0x00 |
| UART0_TX | 0x02 |
| UART0_RX | 0x03 |
| SPICLK0 | 0x04 |
| MISO0 | 0x05 |
| MOSI0 | 0x06 |
| SS0 | 0x07 |
| SPICLK1 | 0x08 |
| MISO1 | 0x09 |
| MOSI1 | 0x0A |
| SS1 | 0x0B |
| SDA0 | 0x0C |
| SCL0 | 0x0D |
| SDA1 | 0x0E |
| SCL1 | 0x0F |
| PWM0 | 0x10 |
| PWM1 | 0x11 |
| PWM2 | 0x12 |
| PWM3 | 0x13 |
| PWM4 | 0x14 |
| PWM5 | 0x15 |
| TIMER_G0 | 0x18 |

Continued on next page

Table  1 – continued from previous page

| Pin | Value |
|---|---|
| TIMER_G1 | 0x19 |
| TIMER_G2 | 0x1A |
| TIMER_G3 | 0x1B |
| TIMER_G4 | 0x1C |
| TIMER_G5 | 0x1D |
| TIMER_G6 | 0x1E |
| TIMER_G7 | 0x1F |
| UART1_TX | 0x22 |
| UART1_RX | 0x23 |
| TIMER_IC0 | 0x38 |
| TIMER_IC1 | 0x39 |
| TIMER_IC2 | 0x3A |
| TIMER_IC3 | 0x3B |
| TIMER_IC4 | 0x3C |
| TIMER_IC5 | 0x3D |
| TIMER_IC6 | 0x3E |
| TIMER_IC7 | 0x3F |

For example:

```
init_io_switch();
set_pin(0, SS0);
set_pin(1, MOSI0);
set_pin(3, SPICLK0);
```

This would connect a SPI interface:

- Pin 0: SS0

- Pin 1: MOSI0

- Pin 2: GPIO

- Pin 3: SPICLK0

- Pin 4: GPIO

- Pin 5: GPIO

- Pin 6: GPIO

- Pin 7: GPIO

### IO Switch Modes and Pin Mapping

Note that the IO switch IP is a customizable IP can be configured by users inside a Vivado project (by double clicking the IP icon of the IO switch). There are 4 pre-defined modes (*pmod*, *dual pmod*, *arduino*, *raspberrypi*) and 1 fully-customizable mode (*custom*) for users to choose. In the base overlay, we have only used *pmod* and *arduino* as the IO switch modes.

Switch mappings used for Pmod:

| Pin | GPIO | UART | PWM | Timer | SPI | IIC | Input-Capture |
|-----|------|------|-----|-------|-----|-----|---------------|
| D0 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | SS0 | | TIMER_IC0 |
| D1 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | MOSI0 | | TIMER_IC0 |
| D2 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | MISO0 | SCL0 | TIMER_IC0 |
| D3 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | SPICLK0 | SDA0 | TIMER_IC0 |
| D4 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | SS0 | | TIMER_IC0 |
| D5 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | MOSI0 | | TIMER_IC0 |
| D6 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | MISO0 | SCL0 | TIMER_IC0 |
| D7 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | SPICLK0 | SDA0 | TIMER_IC0 |

Note:

- PWM0, TIMER_G0, TIMER_IC0 can only be used once on any pin.

- UART0_TX/RX is supported by Pmod, but not implemented in the base overlay.

- SS0, MOSI0, MISO0, SPICLK0 can either be used on top-row (pins D0 - D3) or bottom-row (D4 - D7) but not both.

- SCL0, SDA0 can either be used on to-row (pins D2 - D3) or bottom-row (D6 - D7) but not both.

Switch mappings used for Arduino:

| Pin | GPIO | UART | PWM | Timer | SPI | IIC | Input-Capture |
|-----|------|------|-----|-------|-----|-----|---------------|
| D0 | GPIO | UART0_RX | | | | | |
| D1 | GPIO | UART0_TX | | | | | |
| D2 | GPIO | | | | | | |
| D3 | GPIO | | PWM0 | TIMER_G0 | | | TIMER_IC0 |
| D4 | GPIO | | | TIMER_G6 | | | TIMER_IC6 |
| D5 | GPIO | | PWM1 | TIMER_G1 | | | TIMER_IC1 |
| D6 | GPIO | | PWM2 | TIMER_G2 | | | TIMER_IC2 |
| D7 | GPIO | | | | | | |
| D8 | GPIO | | | TIMER_G7 | | | TIMER_IC7 |
| D9 | GPIO | | PWM3 | TIMER_G3 | | | TIMER_IC3 |
| D10 | GPIO | | PWM4 | TIMER_G4 | SS0 | | TIMER_IC4 |
| D11 | GPIO | | PWM5 | TIMER_G5 | MOSI0 | | TIMER_IC5 |
| D12 | GPIO | | | | MISO0 | | |
| D13 | GPIO | | | | SPICLK0 | | |
| D14/A0 | GPIO | | | | | | |
| D15/A1 | GPIO | | | | | | |
| D16/A2 | GPIO | | | | | | |
| D17/A3 | GPIO | | | | | | |
| D18/A4 | GPIO | | | | | | |
| D19/A5 | GPIO | | | | | | |

Note:

- On Arduino, a dedicated pair of pins are connected to IIC (not going through the IO switch).

Switch mappings used for dual Pmod:

| Pin | GPIO | UART | PWM | Timer | SPI | IIC | Input-Capture |
|-----|------|------|-----|-------|-----|-----|---------------|
| D0 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | SS0 | | TIMER_IC0 |
| D1 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | MOSI0 | | TIMER_IC0 |
| D2 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | MISO0 | SCL0 | TIMER_IC0 |
| D3 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | SPLCLK0 | SDA0 | TIMER_IC0 |
| D4 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | SS0 | | TIMER_IC0 |
| D5 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | MOSI0 | | TIMER_IC0 |
| D6 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | MISO0 | SCL0 | TIMER_IC0 |
| D7 | GPIO | UART0_RX/TX | PWM0 | TIMER_G0 | SPICLK0 | SDA0 | TIMER_IC0 |

| Pin | GPIO | UART | PWM | Timer | SPI | IIC | Input-Capture |
|-----|------|------|-----|-------|-----|-----|---------------|
| D0 | GPIO | UART0_RX/TX | PWM0 | TIMER_G1 | SS1 | | TIMER_IC1 |
| D1 | GPIO | UART0_RX/TX | PWM0 | TIMER_G1 | MOSI1 | | TIMER_IC1 |
| D2 | GPIO | UART0_RX/TX | PWM0 | TIMER_G1 | MISO1 | SCL1 | TIMER_IC1 |
| D3 | GPIO | UART0_RX/TX | PWM0 | TIMER_G1 | SPICLK1 | SDA1 | TIMER_IC1 |
| D4 | GPIO | UART0_RX/TX | PWM0 | TIMER_G1 | SS1 | | TIMER_IC1 |
| D5 | GPIO | UART0_RX/TX | PWM0 | TIMER_G1 | MOSI1 | | TIMER_IC1 |
| D6 | GPIO | UART0_RX/TX | PWM0 | TIMER_G1 | MISO1 | SCL1 | TIMER_IC1 |
| D7 | GPIO | UART0_RX/TX | PWM0 | TIMER_G1 | SPICLK1 | SDA1 | TIMER_IC1 |

Note:

- PWM0, TIMER_G0, TIMER_IC0 can only be used once on any pin of D0 - D7.

- PWM0, TIMER_G1, TIMER_IC1 can only be used once on any pin of D8 - D15.

- SS0, MOSI0, MISO0, SPICLK0 can either be used on top-row (pins D0 - D3) or bottom-row (D4 - D7) but not both.

- SS1, MOSI1, MISO1, SPICLK1 can either be used on top-row (pins D8 - D11) or bottom-row (D12 - D15) but not both.

- SCL0, SDA0 can either be used on to-row (pins D2 - D3) or bottom-row (D6 - D7) but not both.

- SCL1, SDA1 can either be used on to-row (pins D10 - D11) or bottom-row (D14-D15) but not both.

Switch mappings used for Raspberry Pi:

| Pin | GPIO | UART | PWM | Timer | SPI | IIC | Input-Capture |
|---|---|---|---|---|---|---|---|
| GPIO0 | GPIO | | | | | SDA0 | |
| GPIO1 | GPIO | | | | | SCL0 | |
| GPIO2 | GPIO | | | | | SDA1 | |
| GPIO3 | GPIO | | | | | SCL1 | |
| GPIO4 | GPIO | | | | | | |
| GPIO5 | GPIO | | | | | | |
| GPIO6 | GPIO | | | | | | |
| GPIO7 | GPIO | | | | SS0 | | |
| GPIO8 | GPIO | | | | SS0 | | |
| GPIO9 | GPIO | | | | MISO0 | | |
| GPIO10 | GPIO | | | | MOSI0 | | |
| GPIO11 | GPIO | | | | SPICLK0 | | |
| GPIO12 | GPIO | | PWM0 | | | | |
| GPIO13 | GPIO | | PWM1 | | | | |
| GPIO14 | GPIO | UART0_TX | | | | | |
| GPIO15 | GPIO | UART0_RX | | | | | |
| GPIO16 | GPIO | | | | SS1 | | |
| GPIO17 | GPIO | | | | | | |
| GPIO18 | GPIO | | | | | | |
| GPIO19 | GPIO | | | | MISO1 | | |
| GPIO20 | GPIO | | | | MOSI1 | | |
| GPIO21 | GPIO | | | | SPICLK1 | | |
| GPIO22 | GPIO | | | | | | |
| GPIO23 | GPIO | | | | | | |
| GPIO24 | GPIO | | | | | | |
| GPIO25 | GPIO | | | | | | |

Note:

- SPI0 can have up to two Slave Selects (SS's). SS0 can be used to program the functionality for the IO switch.

## PYNQ MicroBlaze Example

## MicroBlaze C Code

Taking Pmod ALS as an example PYNQ MicroBlaze driver (used to control the Pmod light sensor):

`<PYNQ repository>/pynq/lib/pmod/pmod_als/src/pmod_als.c`

First note that the *pynqmb* header files are included.

```
#include "spi.h"
#include "timer.h"
#include "circular_buffer.h"
```

Next, some constants for commands are defined. These values can be chosen properly. The corresponding Python code will send the appropriate command values to control the PYNQ MicroBlaze application.

By convention, 0x0 is reserved for no command (idle, or acknowledged); then PYNQ MicroBlaze commands can be any non-zero value.

```
// MAILBOX_WRITE_CMD
#define READ_SINGLE_VALUE 0x3
#define READ_AND_LOG      0x7
// Log constants
#define LOG_BASE_ADDRESS (MAILBOX_DATA_PTR(4))
#define LOG_ITEM_SIZE sizeof(u32)
#define LOG_CAPACITY  (4000/LOG_ITEM_SIZE)
```

The ALS peripheral has as SPI interface. An SPI variable is defined and accessible to the remaining part of the program.

```
spi device;
```

The user defined function *get_sample()* calls *spi_transfer()* to read data from the device.

```
u32 get_sample(){
   /*
   ALS data is 8-bit in the middle of 16-bit stream.
   Two bytes need to be read, and data extracted.
   */
   u8 raw_data[2];
   spi_transfer(device, NULL, (char*)raw_data, 2);
   return ( ((raw_data[1] & 0xf0) >> 4) + ((raw_data[0] & 0x0f) << 4) );
}
```

In `main()` notice that no IO switch related functions are called; this is because those functions are performed under the hood automatically by *spi_open()*. Also notice this application does not allow the switch configuration to be modified from Python. This means that if you want to use this code with a different pin configuration, the C code must be modified and recompiled.

```
int main(void)
{
   int cmd;
   u16 als_data;
   u32 delay;

   device = spi_open(3, 2, 1, 0);

   // to initialize the device
   get_sample();
```

Next, the `while(1)` loop continually checks the `MAILBOX_CMD_ADDR` for a non-zero command. Once a command is received from Python, the command is decoded, and executed.

```
// Run application
while(1){

   // wait and store valid command
   while((MAILBOX_CMD_ADDR & 0x01)==0);
   cmd = MAILBOX_CMD_ADDR;
```

Taking the first case, reading a single value; `get_sample()` is called and a value returned to the first position (0) of the `MAILBOX_DATA`.

`MAILBOX_CMD_ADDR` is reset to zero to acknowledge to the ARM processor that the operation is complete and data is available in the mailbox.

Remaining code:

```c
    switch(cmd){

      case READ_SINGLE_VALUE:
    // write out reading, reset mailbox
    MAILBOX_DATA(0) = get_sample();
    MAILBOX_CMD_ADDR = 0x0;

    break;

      case READ_AND_LOG:
      // initialize logging variables, reset cmd
      cb_init(&circular_log, LOG_BASE_ADDRESS, LOG_CAPACITY, LOG_ITEM_SIZE);
      delay = MAILBOX_DATA(1);
      MAILBOX_CMD_ADDR = 0x0;

        do{
            als_data = get_sample();
        cb_push_back(&circular_log, &als_data);
        delay_ms(delay);

        } while((MAILBOX_CMD_ADDR & 0x1)== 0);

        break;

      default:
        // reset command
        MAILBOX_CMD_ADDR = 0x0;
        break;
    }
  }
  return(0);
}
```

### Python Code

With the PYNQ MicroBlaze Driver written, the Python class can be built to communicate with that PYNQ MicroBlaze.

`<PYNQ repository>/pynq/lib/pmod/pmod_als.py`

First the Pmod package is imported:

```python
from . import Pmod
```

Then some other constants are defined:

```python
PMOD_ALS_PROGRAM = "pmod_als.bin"
PMOD_ALS_LOG_START = MAILBOX_OFFSET+16
PMOD_ALS_LOG_END = PMOD_ALS_LOG_START+(1000*4)
RESET = 0x1
READ_SINGLE_VALUE = 0x3
READ_AND_LOG = 0x7
```

The MicroBlaze binary file for the PYNQ MicroBlaze is defined. This is the application executable, and will be loaded into the PYNQ MicroBlaze instruction memory.

The ALS class and an initialization method are defined:

```python
class Pmod_ALS(object):

    def __init__(self, mb_info):
```

The initialization function for the module requires the MicroBlaze information. The `__init__` is called when a module is initialized. For example, from Python:

```python
from pynq.lib.pmod import Pmod_ALS
from pynq.lib.pmod import PMODA
als = Pmod_ALS(PMODA)
```

This will create a `Pmod_ALS` instance, and load the MicroBlaze executable (`PMOD_ALS_PROGRAM`) into the instruction memory of the specified PYNQ MicroBlaze.

Since the MicroBlaze information, imported as Pmod constants, can also be extracted as an attribute of the overlay, the following code also works:

```python
from pynq.overlays.base import BaseOverlay
base = BaseOverlay("base.bit")
als = Pmod_ALS(base.PMODA)
```

In the initialization method, an instance of the `Pmod` class is created. This `Pmod` class controls the basic functionalities of the MicroBlaze processor, including reading commands/data, and writing commands/data.

Internally, when the `Pmod` class is initialized, the `run()` call pulls the PYNQ MicroBlaze out of reset. After this, the PYNQ MicroBlaze will be running the `pmod_als.bin` executable.

The `read()` method in the `Pmod_ALS` class will read an ALS sample and return that value to the caller. The following steps demonstrate a Python to MicroBlaze read transaction specific to the `Pmod_ALS` class.

```python
def read(self):
```

First, the command is written to the MicroBlaze shared memory. In this case the value `READ_SINGLE_VALUE` represents a command value. This value is user defined in the Python code, and must match the value the C program expects for the same function.

```python
self.microblaze.write_blocking_command(READ_SINGLE_VALUE)
```

The command is blocking so that Python code will not proceed unless an acknowledgement has been received from the MicroBlaze. Internally, after the PYNQ MicroBlaze has finished its task, it will write `0x0` to clear the command area. The Python code checks this command area (in this case, the Python code constantly checks whether the `0x3` value is still present at the `CMD_OFFSET`).

Once the command is no longer `0x3` (the acknowledge has been received), the result is read from the data area of the shared memory `MAILBOX_OFFSET`.

```python
data = self.microblaze.read_mailbox(0)
return data
```

### 2.6.5 Python-C Integration

In some instances, the performance of Python classes to manage data transfer to an overlay may not be sufficient. Usually this would be determined by implementing the driver in Python, and profiling to determine performance.

A higher performance library can be developed in a lower level language (e.g. C/C++) and optimized for an overlay. The driver functions in the library can be called from Python using CFFI (C Foreign Function Interface).

CFFI provides a simple way to interface with C code from Python. The CFFI package is preinstalled in the PYNQ image. It supports four modes, API and ABI, each with "in-line" or "out-of-line compilation". *Inline* ABI (Application Binary Interface) compatibility mode allows dynamic loading and running of functions from executable modules, and API mode allows building of C extension modules.

The following example taken from http://docs.python-guide.org/en/latest/scenarios/clibs/ shows the ABI inline mode, calling the C function `strlen()` in from Python

C function prototype:

```
size_t strlen(const char*);
```

The C function prototype is passed to `cdef()`, and can be called using `clib`.

```
from cffi import FFI
ffi = FFI()
ffi.cdef("size_t strlen(const char*);")
clib = ffi.dlopen(None)
length = clib.strlen(b"String to be evaluated.")
print("{}".format(length))
```

C functions inside a shared library can be called from Python using the C Foreign Function Interface (CFFI). The shared library can be compiled online using the CFFI from Python, or it can be compiled offline.

For more information on CFFI and shared libraries refer to:

http://cffi.readthedocs.io/en/latest/overview.html

http://www.tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html

## 2.6.6 PYNQ and Asyncio

Interacting with hardware frequently involves waiting for accelerators to complete or stalling for data. Polling is an inefficient way of waiting for data especially in a language like python which can only have one executing thread at once.

The Python asyncio library manages multiple IO-bound tasks asynchronously, thereby avoiding any blocking caused by waiting for responses from slower IO subsystems. Instead, the program can continue to execute other tasks that are ready to run. When the previously-busy tasks are ready to resume, they will be executed in turn, and the cycle is repeated.

In PYNQ real-time tasks are most often implemented using IP blocks in the Programmable Logic (PL). While such tasks are executing in the PL they can raise interrupts on the PS CPUs at any time. Python's asyncio library provides an effective way to manage such events from asynchronous, IO-bound tasks.

The foundation of asyncio in Pynq is the Interrupts class in the *pynq.interrupt Module* which provides an asyncio-style event that can be used for waiting for interrupts to be raised. The video Library, AXI GPIO and the PynqMicroblaze drivers are build on top of the interrupt event to provide coroutines for any functions that might otherwise block.

### Asyncio Fundammentals

The asyncio concurrency framework relies on coroutines, futures, tasks, and an event loop. We will introduce these briefly before demonstrating their use with some introductory examples.

### Coroutines

Coroutines are a new Python language construct. Coroutines introduce two new keywords `await` and `async` the Python syntax. Coroutines are stateful functions whose execution can be paused. This means that they can yield execution, while they wait on some task or event to complete. While suspended, coroutines maintain their state. They are resumed once the outstanding activity is resolved. The await keyword determines the point in the coroutine where it yields control and from which execution will resume.

### Futures

A `future` is an object that acts as a proxy for a result that is initially unknown, usually because the action has not yet completed. The futures concept is essential components in the internals of asyncio: futures encapsulate pending operations so that they can be put in queues, their state of completion can be queried, and their results can be retrieved when ready. They are meant to be instantiated exclusively by the concurrency framework, rather than directly by the user.

### Tasks

Coroutines do not execute directly. Instead, they are wrapped in `tasks` and registered with an asyncio event loop. tasks are a subclass of futures.

### Event Loop

The event loop is responsible for executing all *ready* tasks, polling the status of suspended tasks, and scheduling outstanding tasks.

An event loop runs only one task at a time. It relies on cooperative scheduling. This means that no task interrupts another, and each task yields control to the event loop when its execution is blocked. The result is single-threaded, concurrent code in which the next cycle of the loop does not start until all the event handlers are executed sequentially.

A simple example is shown below. The example defines an coroutine named `wake_up` defined using the new `async def` syntax. Function main creates an asyncio event loop that wraps the wake_up coroutine in a task called called `wake_up_task` and registers the task with the event loop. Within the coroutine, the `await` statement marks the point at which execution is initially suspended, and later resumed. The loop executes the following schedule:

1. Starts executing wake_up_task

2. Suspends wake_up_task and preserves its state

3. Runs asyncio.sleep runs for 1 to 5 seconds

4. Resumes wake_up_task from preserved state

5. Runs to completion using the preserved state

Finally the event loop is closed.

```python
import asyncio
import random
import time

# Coroutine
async def wake_up(delay):
    '''A coroutine that will yield to asyncio.sleep() for a few seconds
       and then resume, having preserved its state while suspended
```

(continues on next page)

```
    '''

    start_time = time.time()
    print(f'The time is: {time.strftime("%I:%M:%S")}')
    print(f"Suspending coroutine 'wake_up' at 'await` statement\n")
    await asyncio.sleep(delay)
    print(f"Resuming coroutine 'wake_up' from 'await` statement")
    end_time = time.time()
    sleep_time = end_time - start_time
    print(f"'wake-up' was suspended for precisely: {sleep_time} seconds")

# Event loop
if __name__ == '__main__':
    delay = random.randint(1,5)
    my_event_loop = asyncio.get_event_loop()
    try:
        print("Creating task for coroutine 'wake_up'\n")
        wake_up_task = my_event_loop.create_task(wake_up(delay))
        my_event_loop.run_until_complete(wake_up_task)
    except RuntimeError as err:
        print (f'{err}' +
                ' - restart the Jupyter kernel to re-run the event loop')
    finally:
        my_event_loop.close()
```

A sample run of the code produces the following output:

```
Creating task for coroutine 'wake_up'

The time is: 11:09:28
Suspending coroutine 'wake_up' at 'await` statement

Resuming coroutine 'wake_up' from 'await` statement
'wake-up' was suspended for precisely: 3.0080409049987793 seconds
```

Any blocking call in event loop should be replaced with a coroutine. If you do not do this, when a blocking call is reached, it will block the rest of the loop.

If you need blocking calls, they should be in separate threads. Compute workloads should also be in separate threads/processes.

### Instances of Asyncio in pynq

Asyncio can be used for managing a variety of potentially blocking operations in the overlay. A coroutine can be run in an event loop and used to wait for an interrupt to fire. Other user functions can also be run in the event loop. If an interrupt is triggered, any coroutines waiting on the corresponding event will be rescheduled. The responsiveness of the interrupt coroutine will depend on how frequently the user code yields control in the loop.

### GPIO Peripherals

User I/O peripherals can trigger interrupts when switches are toggled or buttons are pressed. Both the *Button* and *Switch* classes have a function `wait_for_level` and a coroutine `wait_for_level_async` which block until the corresponding button or switch has the specified value. This follows a convention throughout the pynq package that that coroutines have an `_async` suffix.

As an example, consider an application where each LED will light up when the corresponding button is pressed. First a coroutine specifying this functionality is defined:

```
base = pynq.overlays.base.BaseOverlay('base.bit')

async def button_to_led(number):
    button = base.buttons[number]
    led = base.leds[number]
    while True:
        await button.wait_for_level_async(1)
        led.on()
        await button.wait_for_level_async(0)
        led.off()
```

Next add instances of the coroutine to the default event loop

```
tasks = [asyncio.ensure_future(button_to_led(i) for i in range(4)]
```

Finally, running the event loop will cause the coroutines to be active. This code runs the event loop until an exception is thrown or the user interrupts the process.

```
asyncio.get_event_loop().run_forever()
```

### PynqMicroblaze

The *PynqMicroblaze* class has an `interrupt` member variable which acts like an asyncio.Event with a wait() coroutine and a clear() method. This event is automatically wired to the correct interrupt pin or set to None if interrupts are not available in the loaded overlay.

For example:

```
def __init__(self)
    self.iop = pynq.lib.PynqMicroblaze(mb_info, IOP_EXECUTABLE)
    if self.iop.interrupt is None:
        warn("Interrupts not available in this Overlay")
```

There are two options for running functions from this new IOP wrapper class. The function can be called from an external asyncio event loop (set up elsewhere), or the function can set up its own event loop and then call its asyncio function from the event loop.

### Async Functions

pynq offers both an asyncio coroutine and a blocking function call for all interrupt-driven functions. It is recommended that this should be extended to any user-provided drivers. The blocking function can be used where there is no need to work with asyncio, or as a convenience function to run the event loop until a specified condition.

The following code defines an asyncio coroutine. Notice the `async` and `await` keywords are the only additional code needed to make this function an asyncio coroutine.

```
async def interrupt_handler_async(self, value):
    if self.iop.interrupt is None:
        raise RuntimeError('Interrupts not available in this Overlay')
    while(1):
        await self.iop.interrupt.wait() # Wait for interrupt
```

(continues on next page)

```
        # Do something when an interrupt is received
        self.iop.interrupt.clear()
```

### Event Loops

The following code wraps the asyncio coroutine, adding to the default event loop and running it until the coroutine
completes.

```python
def interrupt_handler(self):

    if self.iop.interrupt is None:
        raise RuntimeError('Interrupts not available in this Overlay')
    loop = asyncio.get_event_loop()
    loop.run_until_complete(asyncio.ensure_future(
        self.interrupt_handler_async()
    ))
```

### Custom Interrupt Handling

The *Interrupts* class allows custom interrupt handlers to be built in Python.

This class abstracts away management of the AXI interrupt controller in the PL. It is not necessary to examine this code
in detail to use interrupts. The interrupt class takes the pin name of the interrupt line and offers a single wait_async
coroutine and the corresponding wait function that wraps it. The interrupt is only enabled in the hardware for as long
as a coroutine is waiting on an *Interrupt* object. The general pattern for using an Interrupt is as follows:

```python
while condition:
    await interrupt.wait()
    # Clear interrupt
```

This pattern avoids race conditions between the interrupt and the controller and ensures that an interrupt isn't seen
multiple times.

### Examples

For more examples, see the AsyncIO Buttons Notebook in the on the Pynq-Z1 in the following directory:

```
<Jupyter Dashboard>/base/board/
```

## 2.6.7 Python Overlay API

The Python API is the user interface for the overlay, exposing the programmable functionality in the design.

An API for a PYNQ overlay can consist of

- a simple Python wrapper that interfaces directly with the hardware IP blocks in the overlay

- a more substantial Python layer utilising other Python packages

- a Python library that interfaces to a lower level higher performance library (written in C/C++ for example) to
  control the overlay

The API for an overlay will manage the transfer of data between the Python environment in the PS, and the overlay in the PL. This may be the transfer of data directly from the PS to a peripheral or managing system memory to allow a peripheral in the PL to read or write data from DRAM that can also be access by from the Python environment.

### The Default API

When and Overlay is loaded using the `pynq.Overlay` function all of the IP and hierarchies in the overlay will have drivers assigned to them and used to construct an object hierarchy. The IP can then be accessed via attributes on the returned overlay class using the names of the IP and hierarchies in the block diagram.

If no driver has been specified for a type of IP then a `DefaultIP` will be instantiated offering `read` and `write` functions to access the IP's address space and named accessors to any interrupts or GPIO pins connected to the IP. Hierarchies likewise will be instances of `DefaultHierarchy` offering access to any sub hierarchies or contained IP. The top-level `DefaultOverlay` also acts just like any other IP.

### Customising Drivers

While the default drivers are useful for getting started with new hardware in a design it is preferable to have a higher level driver for end users to interact with. Each of `DefaultIP`, `DefaultHierarchy` and `DefaultOverlay` can be subclassed and automatically bound to elements of the block diagram. New drivers will only be bound when the overlay is reloaded.

### Creating IP Drivers

All IP drivers should inherit from `DefaultIP` and include a `bindto` class attribute consisting of an array of strings. Each string should be a type of IP that the driver should bind to. It is also strongly recommend to call `super().__init__` in the class's constructor. The type of an IP can be found as the VLNV parameter in Vivado or from the `ip_dict` of the overlay.

A template for an IP driver is as follows:

```python
from pynq import DefaultIP

class MyIP(DefaultIP):
    bindto = ['My IP Type']
    def __init__(self, description):
        super().__init__(description)
```

### Creating Hierarchy Drivers

Hierarchy drivers should inherit from `DefaultHierarchy` and provide a static method `checkhierarchy` that takes a description and returns `True` if the driver can bind to it. Any class that meets these two requirements will be automatically added to a list of drivers tested against each hierarchy in a newly loaded overlay.

A template for a hierarchy driver is as follows:

```python
from pynq import DefaultHierarchy

class MyHierarchy(DefaultHierarchy)
    def __init__(self, description):
        super().__init__(description)
```

(continues on next page)

```python
    @staticmethod
    def checkhierarchy(description):
        return False
```

### Creating Custom Overlay Classes

Finally the class changed from the `DefaultOverlay` to provide a more suitable high-level API or provide overlay-specific initialisation. The overlay loader will look for a python file located alongside the bitstream and TCL files, import it and then call the `Overlay` function.

A template for a custom overlay class is as follows:

```python
from pynq import DefaultOverlay


class MyOverlay(DefaultOverlay):
    def __init__(self, bitfile_name, download):
        super().__init__(bitfile_name, download)

        # Other initialisation


Overlay = MyOverlay
```

### Working with Physically Contiguous Memory

In many applications there is a need for large buffers to be transferred between the PS and PL either using DMA engines or HLS IP with AXI master interfaces. In PYNQ the `allocate` functions provides a mechanism to acquire numpy arrays allocated as to be physically contiguous. The allocate function takes `shape` and `dtype` parameters in a similar way to other numpy construction functions.

```python
from pynq import allocate

matrix1 = allocate(shape=(32,32), dtype='f4')
```

These arrays can either be passed directly to the DMA driver's `transfer` function or they contain a `physical_address` attribute which can be used by custom driver code.

When the array is no longer needed the underlying resources should be freed using the `freebuffer` function. Alternatively a context manager can be used to ensure that the buffer is freed at the end of a scope.

```python
with allocate(shape=(32,32), dtype=np.float32) as matrix2:
    dma.sendchannel.transfer(matrix2)
    dma.recvchannel.transfer(matrix1)
    dma.sendchannel.wait()
    dma.recvchannel.wait()
    matrix1.freebuffer()
```

## 2.6.8 Python Packaging

PYNQ uses pip - the Python Packaging Authority's recommended Python Package Installer to install and deliver custom installations of PYNQ. pip's flexible package delivery model has many useful features.

### Packaging `pynq` for Distribution

Packaging the pynq Python project that pip can use is hugely beneficial, but requires carful thought and project architecture. There are many useful references that provide up-to-date information. For more information about how the pynq library is packaged see the following links:

- Open Sourcing a Python Project The Right way

- How to Package Your Python Code

- Packaging Python Projects

### Delivering Non-Python Files

One extremely useful feature that pip provides is the ability to deliver non-python files. In the PYNQ project this is useful for delivering FPGA binaries (.bit), overlay metadata files (.hwh), PYNQ MicroBlaze binaries (.bin), and Jupyter Notebooks (.ipynb), along side the pynq Python libraries.

An example of using pip's **setup.py** file to provide non-python content is shown below:

```python
from setuptools import setup, find_packages
import subprocess
import sys
import shutil
import new_overlay

setup(
    name = "new_overlay",
    version = new_overlay.__version__,
    url = 'https://github.com/your_github/new_overlay',
    license = 'All rights reserved.',
    author = "Your Name",
    author_email = "your@email.com",
    packages = ['new_overlay'],
    package_data = {
    '' : ['*.bit','*.tcl','*.py','*.so'],
    },
    install_requires=[
        'pynq',
    ],
    dependency_links=['http://github.com/xilinx/PYNQ'],
    description = "New custom overlay for PYNQ-Z1"
)
```

The `package_data` argument specifies which files will be installed as part of the package.

Starting from image v2.5, the official PYNQ Github repository will not version-control the following files anymore:

- overlay files (e.g., *base.bit*, *base.hwh*),

- bsp folders(e.g., *bsp_iop_pmod*)

- MicroBlaze binaries (e.g., *pmod_adc.bin*)

We refrain from keeping tracking of these large files; instead, we rely on the SD build flow to update them automatically in each build. Some side-effects are shown below:

- Users should expect longer SD image building time when users are building the image for the first time. Subsequent builds are much faster.

- Users will no longer be able to pip install directly from the official PYNQ Github repository.

To get those files manually, users can simply run the *build.sh* located at the root of the PYNQ repository (make sure you have the correct version of Xilinx tools beforehand).

Once you have all the files, including the files mentioned above, you can package the entire Github repository into a source distribution package. To do that, run

```
python3 setup.py sdist
```

After this, you will find a tarball in the folder *dist*; for example, *pynq-<release.version>.tar.gz*. This is a source distribution so you can bring it to other boards and install it. From a terminal on a board, installing the pynq Python library is as simple as running:

```
export BOARD=<Board>
export PYNQ_JUPYTER_NOTEBOOKS=<Jupyter-Notebook-Location>
pip3 install pynq-<release.version>.tar.gz
```

After pip finishes installation, the board must be rebooted. If you are on a board with a PYNQ image (OS: pynqlinux), you are done at this point. If you are not on a PYNQ image (other OS), the above *pip3 install* is only for the pynq Python library installation; you also need 2 additional services to be started for pynq to be fully-functional.

- PL server service. (Check <PYNQ-repo>/sdbuild/packages/pynq for more information).

- Jupyter notebook service. (Check <PYNQ-repo>/sdbuild/packages/jupyter/start_jupyter.sh as an example).

### Using `pynq` as a Dependency

One of the most useful features of pip is the ability to *depend* on a project, instead of forking or modifying it.

When designing overlays, the best practice for re-using pynq code is to create a Python project as described above and add pynq as a dependency. A good example of this is the BNN-PYNQ project.

The BNN-PYNQ project is an Overlay that *depends* on pynq but does not modify it. The developers list pynq as a dependency in the pip configuration files, which installs pynq (if it isn't already). After installation, the BNN-PYNQ files are added to the installation: notebooks, overlays, and drivers are installed alongside pynq without modifying or breaking the previous source code.

Needless to say, we highly recommend *depending* on pynq instead of *forking and modifying* pynq. An example of depending on pynq is shown in the code segment from the previous section.

### 2.6.9 Overlay Tutorial

This notebook gives an overview of how the Overlay class should be used efficiently.

The redesigned Overlay class has three main design goals * Allow overlay users to find out what is inside an overlay in a consistent manner * Provide a simple way for developers of new hardware designs to test new IP * Facilitate reuse of IP between Overlays

This tutorial is primarily designed to demonstrate the final two points, walking through the process of interacting with a new IP, developing a driver, and finally building a more complex system from multiple IP blocks. All of the code and block diagrams can be found at [https://github.com/PeterOgden/overlay_tutorial]. For these examples to work copy the contents of the overlays directory into the home directory on the PYNQ-Z1 board.
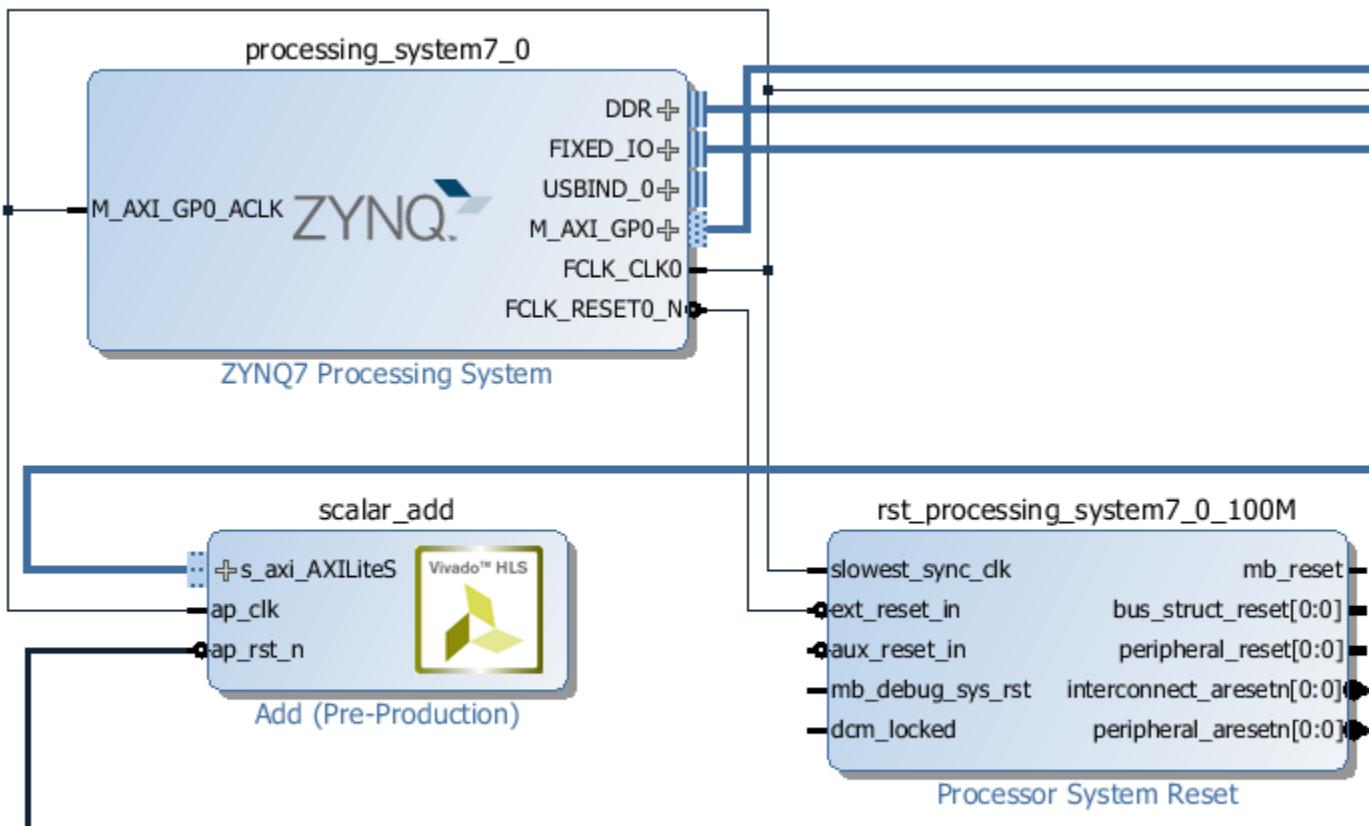
### Developing a Single IP

For this first example we are going to use a simple design with a single IP contained in it. This IP was developed using HLS and adds two 32-bit integers together. The full code for the accelerator is:

```
void add(int a, int b, int& c) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=c

    c = a + b;
}
```

With a block diagram consisting solely of the HLS IP and required glue logic to connect it to the ZYNQ7 IP



To interact with the IP first we need to load the overlay containing the IP.

```
[1]: from pynq import Overlay

overlay = Overlay('/home/xilinx/tutorial_1.bit')
```

Creating the overlay will automatically download it. We can now use a question mark to find out what is in the overlay.

```
[2]: overlay?
```

All of the entries are accessible via attributes on the overlay class with the specified driver. Accessing the

`scalar_add` attribute of the will create a driver for the IP - as there is no driver currently known for the `Add` IP core `DefaultIP` driver will be used so we can interact with IP core.

```
[3]: add_ip = overlay.scalar_add
     add_ip?
```

By providing the HWH file along with overlay we can also expose the register map associated with IP.

```
[4]: add_ip.register_map
```

```
[4]: RegisterMap {
       a = Register(a=0),
       b = Register(b=0),
       c = Register(c=0),
       c_ctrl = Register(c_ap_vld=1, RESERVED=0)
     }
```

We can interact with the IP using the register map directly

```
[5]: add_ip.register_map.a = 3
     add_ip.register_map.b = 4
     add_ip.register_map.c
```

```
[5]: Register(c=7)
```

Alternatively by reading the driver source code generated by HLS we can determine that offsets we need to write the two arguments are at offsets `0x10` and `0x18` and the result can be read back from `0x20`.

```
[6]: add_ip.write(0x10, 4)
     add_ip.write(0x18, 5)
     add_ip.read(0x20)
```

```
[6]: 9
```

### Creating a Driver

While the `UnknownIP` driver is useful for determining that the IP is working it is not the most user-friendly API to expose to the eventual end-users of the overlay. Ideally we want to create an IP-specific driver exposing a single `add` function to call the accelerator. Custom drivers are created by inheriting from `UnknownIP` and adding a `bindto` class attribute consisting of the IP types the driver should bind to. The constructor of the class should take a single `description` parameter and pass it through to the super class `__init__`. The description is a dictionary containing the address map and any interrupts and GPIO pins connected to the IP.

```
[7]: from pynq import DefaultIP

     class AddDriver(DefaultIP):
         def __init__(self, description):
             super().__init__(description=description)

         bindto = ['xilinx.com:hls:add:1.0']

         def add(self, a, b):
             self.write(0x10, a)
             self.write(0x18, b)
             return self.read(0x20)
```

Now if we reload the overlay and query the help again we can see that our new driver is bound to the IP.
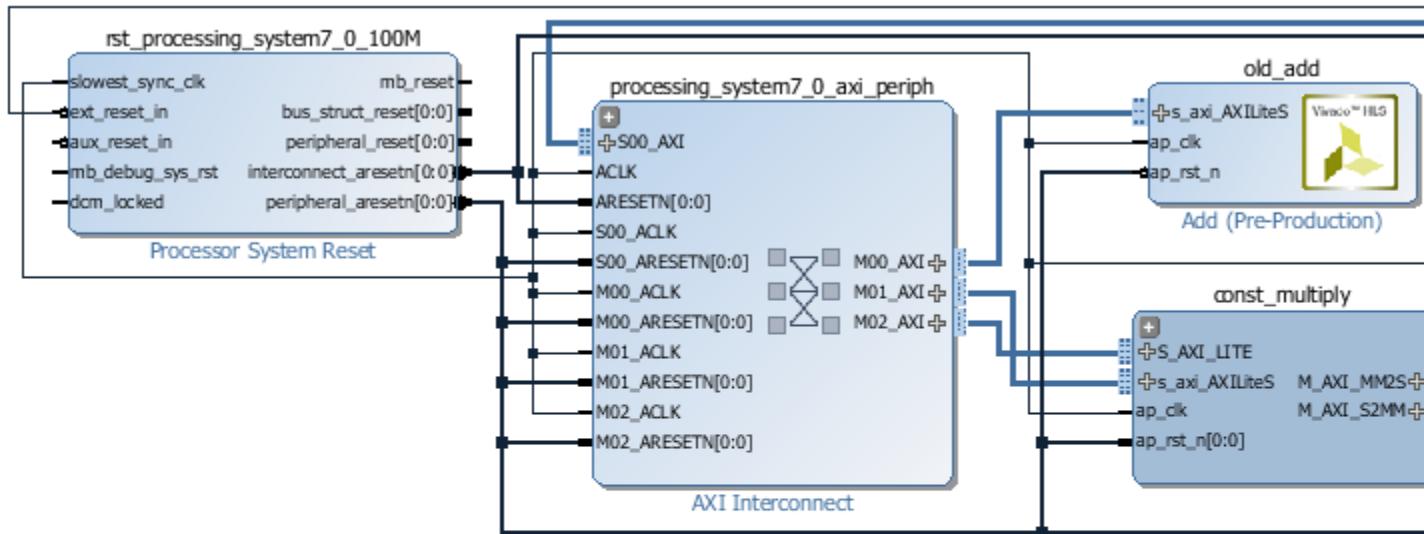
```
[8]: overlay = Overlay('/home/xilinx/tutorial_1.bit')
     overlay?
```

And we can access the same way as before except now our custom driver with an `add` function is created instead of `DefaultIP`

```
[9]: overlay.scalar_add.add(15,20)
```

```
[9]: 35
```

### Reusing IP

Suppose we or someone else develops a new overlay and wants to reuse the existing IP. As long as they import the python file containing the driver class the drivers will be automatically created. As an example consider the next design which, among other things includes a renamed version of the `scalar_add` IP.
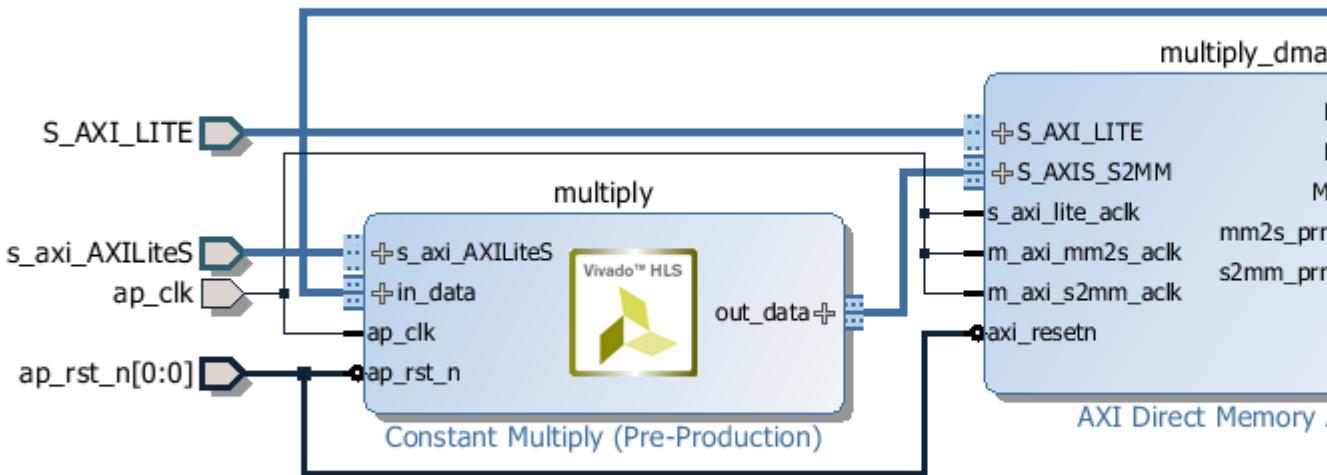


Using the question mark on the new overlay shows that the driver is still bound.

```
[10]: overlay = Overlay('/home/xilinx/tutorial_2.bit')
      overlay?
```

### IP Hierarchies

The block diagram above also contains a hierarchy looking like this:



Containing a custom IP for multiple a stream of numbers by a constant and a DMA engine for transferring the data. As streams are involved and we need correctly handle `TLAST` for the DMA engine the HLS code is a little more complex with additional pragmas and types but the complete code is still relatively short.

```
typedef ap_axiu<32,1,1,1> stream_type;

void mult_constant(stream_type* in_data, stream_type* out_data, ap_int<32> constant) {
#pragma HLS INTERFACE s_axilite register port=constant
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=in_data
#pragma HLS INTERFACE axis port=out_data
    out_data->data = in_data->data * constant;
    out_data->dest = in_data->dest;
    out_data->id = in_data->id;
    out_data->keep = in_data->keep;
    out_data->last = in_data->last;
    out_data->strb = in_data->strb;
    out_data->user = in_data->user;

}
```

Looking at the HLS generated documentation we again discover that to set the constant we need to set the register at offset `0x10` so we can write a simple driver for this purpose

```
[11]: class ConstantMultiplyDriver(DefaultIP):
          def __init__(self, description):
              super().__init__(description=description)

          bindto = ['Xilinx:hls:mult_constant:1.0']

          @property
          def constant(self):
              return self.read(0x10)

          @constant.setter
          def constant(self, value):
              self.write(0x10, value)
```

The DMA engine driver is already included inside the PYNQ driver so nothing special is needed for that other than ensuring the module is imported. Reloading the overlay will make sure that our newly written driver is available for use.

```
[12]: import pynq.lib.dma

      overlay = Overlay('/home/xilinx/tutorial_2.bit')

      dma = overlay.const_multiply.multiply_dma
      multiply = overlay.const_multiply.multiply
```

The DMA driver transfers numpy arrays allocated using `pynq.allocate`. Lets test the system by multiplying 5 numbers by 3.

```
[13]: from pynq import allocate
      import numpy as np

      in_buffer = allocate(shape=(5,), dtype=np.uint32)
      out_buffer = allocate(shape=(5,), dtype=np.uint32)

      for i in range(5):
          in_buffer[i] = i

      multiply.constant = 3
      dma.sendchannel.transfer(in_buffer)
      dma.recvchannel.transfer(out_buffer)
      dma.sendchannel.wait()
      dma.recvchannel.wait()

      out_buffer
```
```
[13]: ContiguousArray([ 0,  3,  6,  9, 12], dtype=uint32)
```

While this is one way to use the IP, it still isn't exactly user-friendly. It would be preferable to treat the entire hierarchy as a single entity and write a driver that hides the implementation details. The overlay class allows for drivers to be written against hierarchies as well as IP but the details are slightly different.

Hierarchy drivers are subclasses of `pynq.DefaultHierarchy` and, similar to `DefaultIP` have a constructor that takes a description of hierarchy. To determine whether the driver should bind to a particular hierarchy the class should also contain a static `checkhierarchy` method which takes the description of a hierarchy and returns `True` if the driver should be bound or `False` if not. Similar to `DefaultIP`, any classes that meet the requirements of subclasses `DefaultHierarchy` and have a `checkhierarchy` method will automatically be registered.

For our constant multiply hierarchy this would look something like:

```
[14]: from pynq import DefaultHierarchy


      class StreamMultiplyDriver(DefaultHierarchy):
          def __init__(self, description):
              super().__init__(description)


          def stream_multiply(self, stream, constant):
              self.multiply.constant = constant
              with allocate(shape=(len(stream),), \
                            dtype=np.uint32) as in_buffer,\
                   allocate(shape=(len(stream),), \
                            dtype=np.uint32) as out_buffer:
                  for i, v, in enumerate(stream):
                      in_buffer[i] = v
                  self.multiply_dma.sendchannel.transfer(in_buffer)
                  self.multiply_dma.recvchannel.transfer(out_buffer)
                  self.multiply_dma.sendchannel.wait()
                  self.multiply_dma.recvchannel.wait()
                  result = out_buffer.copy()
              return result


          @staticmethod
          def checkhierarchy(description):
              if 'multiply_dma' in description['ip'] \
                 and 'multiply' in description['ip']:
                  return True
              return False
```

We can now reload the overlay and ensure the higher-level driver is loaded

```
[15]: overlay = Overlay('/home/xilinx/tutorial_2.bit')
      overlay?
```

and use it

```
[16]: overlay.const_multiply.stream_multiply([1,2,3,4,5], 5)
```

```
[16]: ContiguousArray([ 5, 10, 15, 20, 25], dtype=uint32)
```

### Overlay Customisation

While the default overlay is sufficient for many use cases, some overlays will require more customisation to provide a user-friendly API. As an example the default AXI GPIO drivers expose channels 1 and 2 as separate attributes meaning that accessing the LEDs in the base overlay requires the following contortion

```
[17]: base = Overlay('base.bit')
      base.leds_gpio.channel1[0].on()
```

To mitigate this the overlay developer can provide a custom class for their overlay to expose the subsystems in a more user-friendly way. The base overlay includes custom overlay class which performs the following functions: * Make the AXI GPIO devices better named and range/direction restricted * Make the IOPs accessible through the `pmoda`, `pmodb` and `ardiuno` names * Create a special class to interact with RGB LEDs

The result is that the LEDs can be accessed like:

```
[18]: from pynq.overlays.base import BaseOverlay

      base = BaseOverlay('base.bit')
      base.leds[0].on()
```

Using a well defined class also allows for custom docstrings to be provided also helping end users.

```
[19]: base?
```

### Creating a custom overlay

Custom overlay classes should inherit from `pynq.UnknownOverlay` taking a the full path of the bitstream file and possible additional keyword arguments. These parameters should be passed to `super().__init__()` at the start of `__init__` to initialise the attributes of the Overlay. This example is designed to go with our tutorial_2 overlay and adds a function to more easily call the multiplication function

```
[20]: class TestOverlay(Overlay):
          def __init__(self, bitfile, **kwargs):
              super().__init__(bitfile, **kwargs)

          def multiply(self, stream, constant):
              return self.const_multiply.stream_multiply(stream, constant)
```

To test our new overlay class we can construct it as before.

```
[21]: overlay = TestOverlay('/home/xilinx/tutorial_2.bit')
      overlay.multiply([2,3,4,5,6], 4)
```

```
[21]: ContiguousArray([ 8, 12, 16, 20, 24], dtype=uint32)
```

### Included Drivers

The pynq library includes a number of drivers as part of the `pynq.lib` package. These include

- AXI GPIO
- AXI DMA (simple mode only)
- AXI VDMA
- AXI Interrupt Controller (internal use)
- Pynq-Z1 Audio IP
- Pynq-Z1 HDMI IP
- Color convert IP
- Pixel format conversion
- HDMI input and output frontends
- Pynq Microblaze program loading

## 2.7 PYNQ SD Card

The PYNQ image for supported boards are provided precompiled as downloadable SD card images, so you do not need to rerun this flow for these boards unless you want to make changes to the image flow.

This flow can also be used as a starting point to build a PYNQ image for another Zynq / Zynq Ultrascale board.

The image flow will create the BOOT.bin, the u-boot bootloader, the Linux Device tree blob, and the Linux kernel.

The source files for the PYNQ image flow build can be found here:

```
<PYNQ repository>/sdbuild
```

More details on configuring the root filesystem can be found in the README file in the folder above.

### 2.7.1 Prepare the Building Environment

It is recommended to use a Ubuntu OS to build the image. If you do not have a Ubuntu OS, you may need to prepare a Ubuntu virtual machine (VM) on your host OS. We provide in our repository a *vagrant* file that can help you install the Ubuntu VM on your host OS.

If you do not have a Ubuntu OS, and you need a Ubuntu VM, do the following:

1. Download the vagrant software and the Virtual Box. Install them on your host OS.

2. In your host OS, open a terminal program. Locate your PYNQ repository, where the vagrant file is stored.

   ```
   cd <PYNQ repository>
   ```

3. (optional) Depending on your Virtual Box configurations, you may need to run the following command first; it may help you get better screen resolution for your Ubuntu VM.

   ```
   vagrant plugin install vagrant-vbguest
   ```

4. You can then prepare the VM using the following command. This step will prepare a Ubuntu VM called *pynq_vm* on your Virtual Box. For this VM, username and password are both defaulted to *vagrant*. The Ubuntu packages on the VM will be updated during this process; the Ubuntu desktop will also be installed so you can install Xilinx software later.

   ```
   vagrant up
   ```

   After the VM has been successfully loaded, you will see a folder */pynq* on your VM; this folder is shared with your PYNQ repository on your host OS.

5. (optional) To restart the VM without losing the shared folder, in your terminal, run:

   ```
   vagrant reload
   ```

6. Now you are ready to install Xilinx tools. You will need PetaLinux, Vivado, and SDK for building PYNQ image. Starting from image v2.5, SDx is no longer needed. The version of Xilinx tools for each PYNQ release is shown below:

| Release version | Xilinx Tool Version |
|---|---|
| v1.4 | 2015.4 |
| v2.0 | 2016.1 |
| v2.1 | 2017.4 |
| v2.2 | 2017.4 |
| v2.3 | 2018.2 |
| v2.4 | 2018.3 |
| v2.5 | 2019.1 |

If you already have a Ubuntu OS, you can do the following:

1. Install dependencies using the following script. This is necessary if you are not using our vagrant file to prepare the environment.

```
<PYNQ repository>/sdbuild/scripts/setup_host.sh
```

2. Install correct version of the Xilinx tools, including PetaLinux, Vivado, and SDK. See the above table for the correct version of each release.

## 2.7.2 Building the Image

Once you have the building environment ready, you can start to build the image following the steps below. You don't have to rerun the *setup_host.sh*.

1. Export PATH and source the appropriate settings for PetaLinux, Vivado, and SDK. Suppose you are using Xilinx 2018.3 tools:

```
export PATH="/opt/crosstool-ng/bin:/opt/qemu/bin:$PATH"
source <path-to-vivado>/Vivado/2018.3/settings64.sh
source <path-to-sdk>/SDK/2018.3/settings64.sh
source <path-to-petalinux>/petalinux-v2018.3-final/settings.sh
petalinux-util --webtalk off
```

Currently the SD build flow is checking the PetaLinux path for the version number, so please make sure the version number appears in your installation path of Petalinux. In the above commands, the SD build flow will recognize the Petalinux version number as 2018.3.

2. Navigate to the following directory and run make

```
cd <PYNQ repository>/sdbuild/
make
```

The build flow can take several hours. By default images for all of the supported boards will be built.

## 2.7.3 Retargeting to a Different Board

Additional boards are supported through external *board repositories*. A board repository consists of a directory for each board consisting of a spec file and any other files. The board repository is treated the same way as the `<PYNQ repository>/boards` directory.

### Elements of the specification file

The specification file should be name `<BOARD>.spec` where BOARD is the name of the board directory. A minimal spec file contains the following information

```
ARCH_${BOARD} := arm
BSP_${BOARD} := mybsp.bsp
BITSTREAM_${BOARD} := mybitstream.bsp
```

where `${BOARD}` is also the name of the board. The ARCH should be *arm* for Zynq-7000 or *aarch64* for Zynq UltraScale+. If no bitstream is provided then the one included in the BSP will be used by default. All paths in this file should be relative to the board directory.

To customise the BSP a `petalinux_bsp` folder can be included in the board directory the contents of which will be added to the provided BSP before the project is created. See the ZCU104 for an example of this in action. This is designed to allow for additional drivers, kernel or boot-file patches and device tree configuration that are helpful to support elements of PYNQ to be added to a pre-existing BSP.

If a suitable PetaLinux BSP is unavailable for the board then this can be left blank; in this case, an HDF file needs to be provided in the board directory. The *system.hdf* file should be placed in the `petalinux_bsp/ hardware_project` folder and a new generic BSP will be created as part of the build flow.

### Board-specific packages

A `packages` directory can be included in board directory with the same layout as the `<PYNQ repository>/ sdbuild/packages` directory. Each subdirectory is a package that can optionally be installed as part of image creation. See `<PYNQ repository>/sdbuild/packages/README.md` for a description of the format of a PYNQ sdbuild package.

To add a package to the image you must also define a `STAGE4_PACKAGE_${BOARD}` variable in your spec file. These can either packages in the standard sdbuild library or ones contained within the board package. It is often useful to add the `pynq` package to this list which will ensure that a customised PYNQ installation is included in your final image.

### Using the PYNQ package

The `pynq` package will treat your board directory the same as any of the officially supported boards. This means, in particular, that:

1. A `notebooks` folder, if it exists, will be copied into the `jupyter_notebooks` folder in the image. Notebooks here will overwrite any of the default ones.

2. Any directory containing a bitstream will be treated as an overlay and copied into the overlays folder of the PYNQ installation. Any notebooks will likewise by installed in an overlay-specific subdirectory.

### 2.7.4 Building from a board repository

To build from a third-party board repository pass the `${BOARDDIR}` variable to the sdbuild makefile.

```
cd <PYNQ repository>/sdbuild/
make BOARDDIR=${BOARD_REPO}
```

The board repo should be provided as an absolute path. The `${BOARDDIR}` variable can be combined with the `${BOARD}` variable if the repository contains multiple boards and only a subset should be built.

## 2.8 `pynq` Package

All PYNQ code is contained in the *pynq* Python package and can be found on the on the Github repository.

To learn more about Python package structures, please refer to the official python documentation.

Foundational modules:

- pynq.ps - Facilitates management of the Processing System (PS) and PS/PL interface.

- pynq.pl - Facilitates management of the Programmable Logic (PL).

- pynq.overlay - Manages the state, drivers, and and contents of overlays.

- pynq.bitstream - Instantiates class for PL bitstream (full/partial).

- pynq.devicetree - Instantiates the device tree segment class.

Data Movement modules:

- pynq.mmio - Implements PYNQ Memory Mapped IO (MMIO) API

- pynq.gpio - Implements PYNQ General-Purpose IO (GPIO) by wrapping the Linux Sysfs API

- pynq.xlnk - Implements Contiguous Memory Allocation for PYNQ DMA

- pynq.buffer - Implements a buffer class for DMA engines and accelerators

Additional modules:

- pynq.interrupt - Implements PYNQ asyncio

- pynq.pmbus - PYNQ class for reading power measurements from PMBus

- pynq.uio - Interacts directly with a UIO device

- pynq.registers - Allows users to access registers easily

Sub-packages:

- pynq.lib - Contains sub-packages with drivers for for PMOD, Arduino and Logictools PYNQ Libraries, and drivers for various communication controllers (GPIO, DMA, Video, Audio, etc.)

- pynq.pl_server - Contains sub-packages for PL server to work across multiple devices. It also includes the overlay metadata parsers (e.g., tcl, hwh)

### 2.8.1 pynq.lib Package

pynq.lib contains the arduino, pmod, and logictools subpackages, and additional modules for communicating with other controllers in an overlay.

**Modules:**

- pynq.lib.audio - Implements mono-mode audio driver using pulse density modulation (PDM)

- pynq.lib.axigpio - Implements driver for AXI GPIO IP

- pynq.lib.iic - Implements driver for AXI IIC IP

- pynq.lib.button - Implements driver for AXI GPIO push button IP

- pynq.lib.dma - Implements driver for the AXI Direct Memory Access (DMA) IP

- pynq.lib.led - Implements driver for AXI GPIO LED IP

- pynq.lib.pynqmicroblaze - Implements communcation and control for a PYNQ MicroBlaze subsystem

- pynq.lib.rgbled - Implements driver for AXI GPIO multi color LEDs

- pynq.lib.switch - Implements driver for AXI GPIO Dual In-Line (DIP) switches

- pynq.lib.wifi - Implements driver for WiFi

- pynq.lib.video - Implements driver for HDMI video input/output

**Subpackages:**

- pynq.lib.arduino - Implements driver for Arduino IO Processor Subsystem
- pynq.lib.pmod - Implements driver for PMOD IO Processor Subsystem
- pynq.lib.rpi - Implements driver for Raspberry Pi IO Processor Subsystem
- pynq.lib.logictools - Implements driver for Logictools IP Processor Subsystem

## pynq.lib.arduino Package

The pynq.lib.arduino package is a collection of drivers for controlling peripherals attached to an Arduino pin interface. The Arduino interface can control Arduino peripherals or Grove peripherals (via the PYNQ Grove shield)

## pynq.lib.arduino.arduino_analog Module

**class** `pynq.lib.arduino.arduino_analog.`**`Arduino_Analog`**(*mb_info*, *gr_pin*)

Bases: `object`

This class controls the Arduino Analog.

XADC is an internal analog controller in the hardware. This class provides API to do analog reads from IOP.

**microblaze**

Microblaze processor instance used by this module.

> **Type** Arduino

**log_running**

The state of the log (0: stopped, 1: started).

> **Type** int

**log_interval_ms**

Time in milliseconds between samples on the same channel.

> **Type** int

**gr_pin**

A group of pins on arduino-grove shield.

> **Type** list

**num_channels**

The number of channels sampled.

> **Type** int

**get_log**()

Return list of logged samples.

> **Returns** List of valid voltage samples (floats) from the ADC sensor.

> **Return type** list

**get_log_raw**()

Return list of logged raw samples.

> **Returns** List of valid raw samples from the analog device.

> **Return type** list

**read**()
> Read the voltage value from the analog peripheral.
>
> > **Returns** The float values after translation.
> >
> > **Return type** list

**read_raw**()
> Read the analog raw value from the analog peripheral.
>
> > **Returns** The raw values from the analog device.
> >
> > **Return type** list

**reset**()
> Resets the system monitor for analog devices.
>
> > **Returns**
> >
> > **Return type** None

**set_log_interval_ms**(*log_interval_ms*)
> Set the length of the log for the analog peripheral.
>
> This method can set the time interval between two samples, so that users can read out multiple values in a single log.
>
> > **Parameters** **log_interval_ms** (*int*) – The time between two samples in milliseconds, for logging only.
> >
> > **Returns**
> >
> > **Return type** None

**start_log**()
> Start recording multiple voltage values (float) in a log.
>
> This method will first call set_log_interval_ms() before writing to the MMIO.
>
> > **Returns**
> >
> > **Return type** None

**start_log_raw**()
> Start recording raw data in a log.
>
> This method will first call set_log_interval_ms() before writing to the MMIO.
>
> > **Returns**
> >
> > **Return type** None

**stop_log**()
> Stop recording the voltage values in the log.
>
> This can be done by calling the stop_log_raw() method.
>
> > **Returns**
> >
> > **Return type** None

**stop_log_raw**()
> Stop recording the raw values in the log.
>
> Simply write 0xC to the MMIO to stop the log.
>
> > **Returns**

> **Return type** None

## pynq.lib.arduino.arduino_grove_adc Module

**class** pynq.lib.arduino.arduino_grove_adc.**Grove_ADC**(*mb_info*, *gr_pin*)

> Bases: object

> This class controls the Grove IIC ADC.

> Grove ADC is a 12-bit precision ADC module based on ADC121C021. Hardware version: v1.2.

> **microblaze**
>> Microblaze processor instance used by this module.
>>
>>> **Type** Arduino

> **log_running**
>> The state of the log (0: stopped, 1: started).
>>
>>> **Type** int

> **log_interval_ms**
>> Time in milliseconds between sampled reads.
>>
>>> **Type** int

> **get_log**()
>> Return list of logged samples.
>>
>>> **Returns** List of valid voltage samples (floats) from the ADC sensor.
>>>
>>> **Return type** list

> **get_log_raw**()
>> Return list of logged raw samples.
>>
>>> **Returns** List of valid raw samples from the ADC sensor.
>>>
>>> **Return type** list

> **read**()
>> Read the ADC voltage from the Grove ADC peripheral.
>>
>>> **Returns** The float value after translation.
>>>
>>> **Return type** float

> **read_raw**()
>> Read the ADC raw value from the Grove ADC peripheral.
>>
>>> **Returns** The raw value from the sensor.
>>>
>>> **Return type** int

> **reset**()
>> Resets/initializes the ADC.
>>
>>> **Returns**
>>>
>>> **Return type** None

> **set_log_interval_ms**(*log_interval_ms*)
>> Set the length of the log for the Grove ADC peripheral.

This method can set the time interval between two samples, so that users can read out multiple values in a single log.

> **Parameters** `log_interval_ms` (*int*) – The time between two samples in milliseconds, for logging only.
>
> **Returns**
>
> **Return type** None

**`start_log()`**
Start recording multiple voltage values (float) in a log.

This method will first call set_log_interval_ms() before sending the command.

> **Returns**
>
> **Return type** None

**`start_log_raw()`**
Start recording raw data in a log.

This method will first call set_log_interval_ms() before sending the command.

> **Returns**
>
> **Return type** None

**`stop_log()`**
Stop recording the voltage values in the log.

Simply send the command 0xC to stop the log.

> **Returns**
>
> **Return type** None

**`stop_log_raw()`**
Stop recording the raw values in the log.

Simply send the command 0xC to stop the log.

> **Returns**
>
> **Return type** None

## pynq.lib.arduino.arduino_grove_buzzer Module

**`class`** `pynq.lib.arduino.arduino_grove_buzzer.`**`Grove_Buzzer`**(*mb_info*, *gr_pin*)
Bases: `object`

This class controls the Grove Buzzer.

The grove buzzer module has a piezo buzzer as the main component. The piezo can be connected to digital outputs, and will emit a tone when the output is HIGH. Alternatively, it can be connected to an analog pulse-width modulation output to generate various tones and effects. Hardware version: v1.2.

**`microblaze`**
Microblaze processor instance used by this module.

> **Type** Arduino

**`play_melody()`**
Play a melody.

> **Returns**
>
> **Return type** None

**play_tone**(*tone_period*, *num_cycles*)

> Play a single tone with tone_period for num_cycles
>
> > **Parameters**
> >
> > - **tone_period** (*int*) – The period of the tone in microsecond.
> >
> > - **num_cycles** (*int*) – The number of cycles for the tone to be played.
> >
> > **Returns**
> >
> > **Return type** None

## pynq.lib.arduino.arduino_grove_ear_hr Module

**class** pynq.lib.arduino.arduino_grove_ear_hr.**Grove_EarHR**(*mb_info*, *gr_pin*)

> Bases: object
>
> This class controls the Grove ear clip heart rate sensor. Sensor model: MED03212P.
>
> **microblaze**
>
> > Microblaze processor instance used by this module.
> >
> > > **Type** Arduino
>
> **read**()
>
> > Read the heart rate from the sensor.
> >
> > > **Returns** The heart rate as beats per minute
> > >
> > > **Return type** float
>
> **read_raw**()
>
> > Read the number of heart beats.
> >
> > Read the number of beats since the sensor initialization; also read the time elapsed in ms between the latest two heart beats.
> >
> > > **Returns** Number of heart beats and the time elapsed between 2 latest beats.
> > >
> > > **Return type** tuple

## pynq.lib.arduino.arduino_grove_finger_hr Module

**class** pynq.lib.arduino.arduino_grove_finger_hr.**Grove_FingerHR**(*mb_info*, *gr_pin*)

> Bases: object
>
> This class controls the Grove finger clip heart rate sensor.
>
> Grove Finger sensor based on the TCS3414CS. Hardware version: v1.3.
>
> **microblaze**
>
> > Microblaze processor instance used by this module.
> >
> > > **Type** Arduino
>
> **log_running**
>
> > The state of the log (0: stopped, 1: started).

> > **Type** int

**log_interval_ms**
> Time in milliseconds between sampled reads.

> > **Type** int

**get_log**()
> Return list of logged samples.

> > **Returns** List of integers containing the heart rate.

> > **Return type** list

**read**()
> Read the heart rate value from the Grove Finger HR peripheral.

> > **Returns** An integer representing the heart rate frequency.

> > **Return type** int

**start_log**(*log_interval_ms=100*)
> Start recording multiple heart rate values in a log.

> This method will first call set the log interval before writing to the MMIO.

> > **Parameters** **log_interval_ms** (*int*) – The time between two samples in milliseconds.

> > **Returns**

> > **Return type** None

**stop_log**()
> Stop recording the values in the log.

> Simply write 0xC to the MMIO to stop the log.

> > **Returns**

> > **Return type** None

### pynq.lib.arduino.arduino_grove_haptic_motor Module

**class** pynq.lib.arduino.arduino_grove_haptic_motor.**Grove_HapticMotor**(*mb_info*,
> *gr_pin*)

> Bases: `object`

> This class controls the Grove Haptic Motor based on the DRV2605L. Hardware version v0.9.

> **microblaze**
> > Microblaze processor instance used by this module.

> > > **Type** Arduino

> **is_playing**()
> > Check if a vibration effect is running on the motor.

> > > **Returns** True if a vibration effect is playing, false otherwise

> > > **Return type** bool

> **play**(*effect*)
> > Play a vibration effect on the Grove Haptic Motor peripheral.

> > Valid effect identifiers are in the range [1, 127].

---

> **Parameters effect** (*int*) – An integer that specifies the effect.
>
> **Returns**
>
> **Return type** None

**play_sequence**(*sequence*)

Play a sequence of effects possibly separated by pauses.

At most 8 effects or pauses can be specified at a time. Pauses are defined using negative integer values in the range [-1, -127] that correspond to a pause length in the range [10, 1270] ms

Valid effect identifiers are in the range [1, 127]

As an example, in the following sequence example: [4,-20,5] effect 4 is played and after a pause of 200 ms effect 5 is played

> **Parameters sequence** (*list*) – At most 8 values specifying effects and pauses.
>
> **Returns**
>
> **Return type** None

**stop**()

Stop an effect or a sequence on the motor peripheral.

> **Returns**
>
> **Return type** None

## pynq.lib.arduino.arduino_grove_imu Module

**class** pynq.lib.arduino.arduino_grove_imu.**Grove_IMU**(*mb_info*, *gr_pin*)

Bases: object

This class controls the Grove IIC IMU.

Grove IMU 10DOF is a combination of grove IMU 9DOF (MPU9250) and grove barometer sensor (BMP180). MPU-9250 is a 9-axis motion tracking device that combines a 3-axis gyroscope, 3-axis accelerometer, 3-axis magnetometer and a Digital Motion Processor (DMP). BMP180 is a high precision, low power digital pressure sensor. Hardware version: v1.1.

**microblaze**

Microblaze processor instance used by this module.

> **Type** Arduino

**get_accl**()

Get the data from the accelerometer.

> **Returns** A list of the acceleration data along X-axis, Y-axis, and Z-axis.
>
> **Return type** list

**get_altitude**()

Get the current altitude.

> **Returns** The altitude value.
>
> **Return type** float

**get_atm**()

Get the current pressure in relative atmosphere.

> **Returns** The related atmosphere.

> > > **Return type** float

**get_compass()**
> Get the data from the magnetometer.

> > **Returns** A list of the compass data along X-axis, Y-axis, and Z-axis.

> > **Return type** list

**get_gyro()**
> Get the data from the gyroscope.

> > **Returns** A list of the gyro data along X-axis, Y-axis, and Z-axis.

> > **Return type** list

**get_heading()**
> Get the value of the heading.

> > **Returns** The angle deviated from the X-axis, toward the positive Y-axis.

> > **Return type** float

**get_pressure()**
> Get the current pressure in Pa.

> > **Returns** The pressure value.

> > **Return type** float

**get_temperature()**
> Get the current temperature in degree C.

> > **Returns** The temperature value.

> > **Return type** float

**get_tilt_heading()**
> Get the value of the tilt heading.

> > **Returns** The tilt heading value.

> > **Return type** float

**reset()**
> Reset all the sensors on the grove IMU.

> > **Returns**

> > **Return type** None

## pynq.lib.arduino.arduino_grove_ledbar Module

**class** `pynq.lib.arduino.arduino_grove_ledbar.`**`Grove_LEDbar`**(*mb_info*, *gr_pin*)
> Bases: `object`

> This class controls the Grove LED BAR.

> Grove LED Bar is comprised of a 10 segment LED gauge bar and an MY9221 LED controlling chip. Model: LED05031P. Hardware version: v2.0.

> **microblaze**
> > Microblaze processor instance used by this module.

> > **Type** Arduino

**read**()
> Reads the current status of LEDbar.

> Reads the current status of LED bar and returns 10-bit binary string. Each bit position corresponds to a LED position in the LEDbar, and bit value corresponds to the LED state.

> Red LED corresponds to the LSB, while green LED corresponds to the MSB.

>> **Returns** String of 10 binary bits.

>> **Return type** str

**reset**()
> Resets the LEDbar.

> Clears the LED bar, sets all LEDs to OFF state.

>> **Returns**

>> **Return type** None

**write_binary**(*data_in*)
> Set individual LEDs in the LEDbar based on 10 bit binary input.

> Each bit in the 10-bit *data_in* points to a LED position on the LEDbar. Red LED corresponds to the LSB, while green LED corresponds to the MSB.

>> **Parameters** **data_in** (*int*) – 10 LSBs of this parameter control the LEDbar.

>> **Returns**

>> **Return type** None

**write_brightness**(*data_in, brightness=[170, 170, 170, 170, 170, 170, 170, 170, 170, 170]*)
> Set individual LEDs with 3 level brightness control.

> Each bit in the 10-bit *data_in* points to a LED position on the LEDbar. Red LED corresponds to the LSB, while green LED corresponds to the MSB.

> Brightness of each LED is controlled by the brightness parameter. There are 3 perceivable levels of brightness: 0xFF : HIGH 0xAA : MED 0x01 : LOW

>> **Parameters**

>>> • **data_in** (*int*) – 10 LSBs of this parameter control the LEDbar.

>>> • **brightness** (*list*) – Each element controls a single LED.

>> **Returns**

>> **Return type** None

**write_level**(*level, bright_level, green_to_red*)
> Set the level to which the leds are to be lit in levels 1 - 10.

> Level can be set in both directions. *set_level* operates by setting all LEDs to the same brightness level.

> There are 4 preset brightness levels: bright_level = 0: off bright_level = 1: low bright_level = 2: medium bright_level = 3: maximum

> *green_to_red* indicates the direction, either from red to green when it is 0, or green to red when it is 1.

>> **Parameters**

>>> • **level** (*int*) – 10 levels exist, where 1 is minimum and 10 is maximum.

>>> • **bright_level** (*int*) – Controls brightness of all LEDs in the LEDbar, from 0 to 3.

---

- **green_to_red**(*int*) – Sets the direction of the sequence.

    **Returns**

    **Return type** None

## pynq.lib.arduino.arduino_grove_light Module

**class** pynq.lib.arduino.arduino_grove_light.**Grove_Light**(*mb_info*, *gr_pin*)

    Bases: *pynq.lib.arduino.arduino_grove_adc.Grove_ADC*

This class controls the grove light sensor.

This class inherits from the Grove_ADC class. To use this module, grove ADC has to be used as a bridge. The light sensor incorporates a Light Dependent Resistor (LDR) GL5528. Hardware version: v1.1.

**microblaze**

    Microblaze processor instance used by this module.

        **Type** Arduino

**log_running**

    The state of the log (0: stopped, 1: started).

        **Type** int

**log_interval_ms**

    Time in milliseconds between sampled reads.

        **Type** int

**get_log**()

    Return list of logged light sensor resistances.

        **Returns** List of valid light sensor resistances.

        **Return type** list

**read**()

    Read the light sensor resistance in from the light sensor.

    This method overrides the definition in grove ADC.

        **Returns** The light reading in terms of the sensor resistance.

        **Return type** float

**start_log**()

    Start recording the light sensor resistance in a log.

    This method will call the start_log_raw() in the parent class.

        **Returns**

        **Return type** None

**stop_log**()

    Stop recording light values in a log.

    This method will call the stop_log_raw() in the parent class.

        **Returns**

        **Return type** None

### pynq.lib.arduino.arduino_grove_oled Module

**class** pynq.lib.arduino.arduino_grove_oled.**Grove_OLED**(*mb_info*, *gr_pin*)
    Bases: object

    This class controls the Grove IIC OLED.

    Grove LED 128x64 Display module is an OLED monochrome 128x64 matrix display module. Model: OLE35046P. Hardware version: v1.1.

    **microblaze**
        Microblaze processor instance used by this module.

            **Type** Arduino

    **clear**()
        Clear the OLED screen.

        This is done by writing empty strings into the OLED in Microblaze.

            **Returns**

            **Return type** None

    **set_contrast**(*brightness*)
        Set the contrast level for the OLED display.

        The contrast level is in [0, 255].

            **Parameters brightness** (*int*) – The brightness of the display.

            **Returns**

            **Return type** None

    **set_horizontal_mode**()
        Set the display mode to horizontal.

            **Returns**

            **Return type** None

    **set_inverse_mode**()
        Set the display mode to inverse.

            **Returns**

            **Return type** None

    **set_normal_mode**()
        Set the display mode to normal.

            **Returns**

            **Return type** None

    **set_page_mode**()
        Set the display mode to paged.

            **Returns**

            **Return type** None

    **set_position**(*row*, *column*)
        Set the position of the display.

        The position is indicated by (row, column).

> > **Parameters**
> >
> > - **row** (*int*) – The row number to start the display.
> >
> > - **column** (*int*) – The column number to start the display.
> >
> > **Returns**
> >
> > **Return type** None

> **write**(*text*)
>
> > Write a new text string on the OLED.
> >
> > Clear the screen first to correctly show the new text.
> >
> > > **Parameters** **text** (*str*) – The text string to be displayed on the OLED screen.
> > >
> > > **Returns**
> > >
> > > **Return type** None

## pynq.lib.arduino.arduino_grove_pir Module

**class** pynq.lib.arduino.arduino_grove_pir.**Grove_PIR**(*mb_info*, *gr_pin*)

> Bases: *pynq.lib.arduino.arduino_io.Arduino_IO*
>
> This class controls the PIR motion sensor.
>
> Hardware version: v1.2.
>
> **microblaze**
>
> > Microblaze processor instance used by this module.
> >
> > > **Type** Arduino
>
> **read**()
>
> > Receive the value from the PIR sensor.
> >
> > Returns 0 when there is no motion, and returns 1 otherwise.
> >
> > > **Returns** The data (0 or 1) read from the PIR sensor.
> > >
> > > **Return type** int

## pynq.lib.arduino.arduino_grove_th02 Module

**class** pynq.lib.arduino.arduino_grove_th02.**Grove_TH02**(*mb_info*, *gr_pin*)

> Bases: object
>
> This class controls the Grove I2C Temperature and Humidity sensor.
>
> Temperature & humidity sensor (high-accuracy & mini). Hardware version: v1.0.
>
> **microblaze**
>
> > Microblaze processor instance used by this module.
> >
> > > **Type** Arduino
>
> **log_running**
>
> > The state of the log (0: stopped, 1: started).
> >
> > > **Type** int

**log_interval_ms**
> Time in milliseconds between sampled reads.

>> **Type** int

**get_log**()
> Return list of logged samples.

>> **Returns** List of tuples containing (temperature, humidity)

>> **Return type** list

**read**()
> Read the temperature and humidity values from the TH02 peripheral.

>> **Returns** Tuple containing (temperature, humidity)

>> **Return type** tuple

**start_log**(*log_interval_ms=100*)
> Start recording multiple heart rate values in a log.

> This method will first call set the log interval before sending the command.

>> **Parameters** **log_interval_ms** (*int*) – The time between two samples in milliseconds.

>> **Returns**

>> **Return type** None

**stop_log**()
> Stop recording the values in the log.

> Simply send the command 0xC to stop the log.

>> **Returns**

>> **Return type** None

### pynq.lib.arduino.arduino_grove_tmp Module

**class** pynq.lib.arduino.arduino_grove_tmp.**Grove_TMP**(*mb_info*, *gr_pin*, *version='v1.2'*)
> Bases: *pynq.lib.arduino.arduino_grove_adc.Grove_ADC*

> This class controls the grove temperature sensor.

> This class inherits from the Grove_ADC class. To use this module, grove ADC has to be used as a bridge. The temperature sensor uses a thermistor to detect the ambient temperature. Hardware version: v1.2.

**microblaze**
> Microblaze processor instance used by this module.

>> **Type** Arduino

**log_running**
> The state of the log (0: stopped, 1: started).

>> **Type** int

**log_interval_ms**
> Time in milliseconds between sampled reads.

>> **Type** int

**bValue**
>   The thermistor constant.
>
>>   **Type** int

**get_log()**
>   Return list of logged temperature samples.
>
>>   **Returns** List of valid temperature readings from the temperature sensor.
>>
>>   **Return type** list

**read()**
>   Read temperature values in Celsius from temperature sensor.
>
>   This method overrides the definition in Grove_ADC.
>
>>   **Returns** The temperature reading in Celsius.
>>
>>   **Return type** float

**start_log()**
>   Start recording temperature in a log.
>
>   This method will call the start_log_raw() in the parent class.

**stop_log()**
>   Stop recording temperature in a log.
>
>   This method will call the stop_log_raw() in the parent class.
>
>>   **Returns**
>>
>>   **Return type** None

## pynq.lib.arduino.arduino_io Module

**class** pynq.lib.arduino.arduino_io.**Arduino_IO**(*mb_info*, *index*, *direction*)
>   Bases: object
>
>   This class controls the Arduino IO pins as inputs or outputs.

---

**Note:** The parameter 'direction' determines whether the instance is input/output: 'in' : receiving input from offchip to onchip. 'out' : sending output from onchip to offchip.

---

---

**Note:** The index of the Arduino pins: upper row, from right to left: {0, 1, ..., 13}. (D0 - D13) lower row, from left to right: {14, 15,..., 19}. (A0 - A5)

---

**microblaze**
>   Microblaze processor instance used by this module.
>
>>   **Type** Arduino

**index**
>   The index of the Arduino pin, from 0 to 19.
>
>>   **Type** int

**direction**
>   Input 'in' or output 'out'.

> **Type** str

**read**()
> Receive the value from the offboard Arduino IO device.

---

> **Note:** Only use this function when direction is 'in'.

---

> > **Returns** The data (0 or 1) on the specified Arduino IO pin.
> >
> > **Return type** int

**singleton_instance = None**

**write**(*value*)
> Send the value to the offboard Arduino IO device.

---

> **Note:** Only use this function when direction is 'out'.

---

> > **Parameters** **value** (*int*) – The value to be written to the Arduino IO device.
> >
> > **Returns**
> >
> > **Return type** None

## pynq.lib.arduino.arduino_lcd18 Module

**class** pynq.lib.arduino.arduino_lcd18.**Arduino_LCD18**(*mb_info*)
> Bases: object

This class controls the Adafruit 1.8" LCD shield from AdaFruit.

The LCD panel consists of ST7735 LCD controller, a joystick, and a microSD socket. This class uses the LCD panel (128x160 pixels) and the joystick. The joystick uses A3 analog channel. https://www.adafruit.com/product/802.

**microblaze**
> Microblaze processor instance used by this module.

> > **Type** Arduino

**buf_manager**
> DDR management unit that provides the physical address of the image.

> > **Type** *Xlnk*

**clear**()
> Clear the screen.

> > **Returns**
> >
> > **Return type** None

**display**(*img_path*, *x_pos=0*, *y_pos=127*, *orientation=3*, *background=None*, *frames=1*)
> Animate the image at the desired location for multiple frames.

The maximum screen resolution is 160x128.

Users can specify the position to display the image. For example, to display the image in the center, *x_pos* can be (160-*width*/2), *y_pos* can be (128/2)+(*height*/2).

A typical orientation is 3. The origin of orientation 0, 1, 2, and 3 corresponds to upper right corner, lower right corner, lower left corner, and upper left corner, respectively. Currently, only 1 and 3 are valid orientations. If users choose orientation 1, the picture will be shown upside-down. If users choose orientation 3, the picture will be shown consistently with the LCD screen orientation.

Parameter *background* specifies the color of the background; it is a list of 3 elements: R, G, and B, each with 8 bits for color level.

> **Parameters**
>
> - **img_path** (`str`) – The file path to the image stored in the file system.
> - **x_pos** (`int`) – x position of a pixel where the image starts.
> - **y_pos** (`int`) – y position of a pixel where the image starts.
> - **background** (`list`) – A list of [R, G, B] components for background, each of 8 bits.
> - **orientation** (`int`) – orientation of the image; valid values are 1 and 3.
> - **frames** (`int`) – Number of frames the image is moved, must be less than 65536.
>
> **Returns**
>
> **Return type**  None

**display_async**(*img_path*, *x_pos=0*, *y_pos=127*, *orientation=3*, *background=None*, *frames=1*)
Animate the image at the desired location for multiple frames.

The maximum screen resolution is 160x128.

Users can specify the position to display the image. For example, to display the image in the center, *x_pos* can be (160-*width*/2), *y_pos* can be (128/2)+(*height*/2).

A typical orientation is 3. The origin of orientation 0, 1, 2, and 3 corresponds to upper right corner, lower right corner, lower left corner, and upper left corner, respectively. Currently, only 1 and 3 are valid orientations. If users choose orientation 1, the picture will be shown upside-down. If users choose orientation 3, the picture will be shown consistently with the LCD screen orientation.

Parameter *background* specifies the color of the background; it is a list of 3 elements: R, G, and B, each with 8 bits for color level.

> **Parameters**
>
> - **img_path** (`str`) – The file path to the image stored in the file system.
> - **x_pos** (`int`) – x position of a pixel where the image starts.
> - **y_pos** (`int`) – y position of a pixel where the image starts.
> - **background** (`list`) – A list of [R, G, B] components for background, each of 8 bits.
> - **orientation** (`int`) – orientation of the image; valid values are 1 and 3.
> - **frames** (`int`) – Number of frames the image is moved, must be less than 65536.
>
> **Returns**
>
> **Return type**  None

**draw_filled_rectangle**(*x_start_pos*, *y_start_pos*, *width*, *height*, *color=None*, *background=None*, *orientation=3*)
Draw a filled rectangle.

Parameter *color* specifies the color of the text; it is a list of 3 elements: R, G, and B, each with 8 bits for color level.

Parameter *background* is similar to parameter *color*, except that it specifies the background color.

A typical orientation is 3. The origin of orientation 0, 1, 2, and 3 corresponds to upper right corner, lower right corner, lower left corner, and upper left corner, respectively. Currently, only 1 and 3 are valid orientations. If users choose orientation 1, the picture will be shown upside-down. If users choose orientation 3, the picture will be shown consistently with the LCD screen orientation.

> **Parameters**
> - **x_start_pos** (*int*) – x position (in pixels) where the rectangle starts.
> - **y_start_pos** (*int*) – y position (in pixels) where the rectangle starts.
> - **width** (*int*) – Width of the rectangle (in pixels).
> - **height** (*int*) – Height of the rectangle (in pixels).
> - **color** (*list*) – A list of [R, G, B] components for line color, each of 8 bits.
> - **background** (*list*) – A list of [R, G, B] components for background, each of 8 bits.
> - **orientation** (*int*) – orientation of the image; valid values are 1 and 3.
>
> **Returns**
>
> **Return type** None

**draw_line**(*x_start_pos*, *y_start_pos*, *x_end_pos*, *y_end_pos*, *color=None*, *background=None*, *orientation=3*)
Draw a line from starting point to ending point.

The maximum screen resolution is 160x128.

Parameter *color* specifies the color of the line; it is a list of 3 elements: R, G, and B, each with 8 bits for color level.

Parameter *background* is similar to parameter *color*, except that it specifies the background color.

A typical orientation is 3. The origin of orientation 0, 1, 2, and 3 corresponds to upper right corner, lower right corner, lower left corner, and upper left corner, respectively. Currently, only 1 and 3 are valid orientations. If users choose orientation 1, the picture will be shown upside-down. If users choose orientation 3, the picture will be shown consistently with the LCD screen orientation.

> **Parameters**
> - **x_start_pos** (*int*) – x position (in pixels) where the line starts.
> - **y_start_pos** (*int*) – y position (in pixels) where the line starts.
> - **x_end_pos** (*int*) – x position (in pixels ) where the line ends.
> - **y_end_pos** (*int*) – y position (in pixels) where the line ends.
> - **color** (*list*) – A list of [R, G, B] components for line color, each of 8 bits.
> - **background** (*list*) – A list of [R, G, B] components for background, each of 8 bits.
> - **orientation** (*int*) – orientation of the image; valid values are 1 and 3.
>
> **Returns**
>
> **Return type** None

**print_string** (*x_start_pos*, *y_start_pos*, *text*, *color=None*, *background=None*, *orientation=3*)
    Draw a character with a specific color.

    The maximum screen resolution is 160x128.

    Parameter *color* specifies the color of the text; it is a list of 3 elements: R, G, and B, each with 8 bits for color level.

    Parameter *background* is similar to parameter *color*, except that it specifies the background color.

    A typical orientation is 3. The origin of orientation 0, 1, 2, and 3 corresponds to upper right corner, lower right corner, lower left corner, and upper left corner, respectively. Currently, only 1 and 3 are valid orientations. If users choose orientation 1, the picture will be shown upside-down. If users choose orientation 3, the picture will be shown consistently with the LCD screen orientation.

        **Parameters**

- **x_start_pos** (*int*) – x position (in pixels) where the line starts.
- **y_start_pos** (*int*) – y position (in pixels) where the line starts.
- **text** (*str*) – printable ASCII characters.
- **color** (*list*) – A list of [R, G, B] components for line color, each of 8 bits.
- **background** (*list*) – A list of [R, G, B] components for background, each of 8 bits.
- **orientation** (*int*) – orientation of the image; valid values are 1 and 3.

        **Returns**

        **Return type** None

**read_joystick** ()
    Read the joystick values.

    The joystick values can be read when user is pressing the button toward a specific direction.

    The returned values can be: 1: left; 2: down; 3: center; 4: right; 5: up; 0: no button pressed.

        **Returns** Indicating the direction towards which the button is pushed.

        **Return type** int

## pynq.lib.audio Module

The pynq.lib.audio module is a driver for reading and recording values from an on-board audio mirophone, loading preexisting audio files, or playing audio input to an output device.

**class** pynq.lib.audio.**AudioADAU1761** (*description*)
    Bases: *pynq.overlay.DefaultIP*

    Class to interact with audio codec controller.

    Each raw audio sample is a 24 bits, padded to 32 bits. The audio controller supports both mono and stereo modes, and I2S format of data.

    **buffer**
        The numpy array to store the audio.

        **Type** numpy.ndarray

    **sample_rate**
        Sample rate of the codec.

        **Type** int

> **sample_len**
>> Sample length of the current buffer content.
>>
>>> **Type** int

> **iic_index**
>> The index of the IIC instance in /dev.
>>
>>> **Type** int

> **uio_index**
>> The index of the UIO instance in /dev.
>>
>>> **Type** int

> **bindto = ['xilinx.com:user:audio_codec_ctrl:1.0']**

> **bypass**(*seconds*)
>> Stream audio controller input directly to output.
>>
>> It will run for a certain number of seconds, then stop automatically.
>>
>>> **Parameters** **seconds** (*float*) – The number of seconds to be recorded.
>>>
>>> **Returns**
>>>
>>> **Return type** None

> **configure**(*sample_rate=48000*, *iic_index=1*, *uio_name='audio-codec-ctrl'*)
>> Configure the audio codec.
>>
>> The sample rate of the codec is 48KHz, by default. This method will configure the PLL and codec registers.
>>
>> The parameter *iic_index* is required as input; *uio_index* is calculated automatically from *uio_name*.
>>
>> Users can also explicitly call this function to reconfigure the driver.
>>
>>> **Parameters**
>>>
>>> - **sample_rate** (*int*) – Sample rate of the codec.
>>> - **iic_index** (*int*) – The index of the IIC instance in /dev.
>>> - **uio_name** (*int*) – The name of the UIO configured in the device tree.

> **deselect_inputs**()
>> Deselect the inputs.
>>
>> This method will disable both LINE_IN and MIC inputs.

> **static info**(*file*)
>> Prints information about the sound files.
>>
>> The information includes name, channels, samples, frames, etc.
>>
>> ---
>>
>> **Note:** The file will be searched in the specified path, or in the working directory in case the path does not exist.
>>
>> ---
>>
>>> **Parameters** **file** (*string*) – File name, with a default extension of *wav*.
>>>
>>> **Returns**
>>>
>>> **Return type** None

**load**(*file*)

Loads file into internal audio buffer.

The recorded file is of format *.wav*. Note that we expect 32-bit samples in the buffer while the each saved sample is only 24 bits. Hence we need to pad the highest 8 bits when reading the wave file.

---

**Note:** The file will be searched in the specified path, or in the working directory in case the path does not exist.

---

> **Parameters** **file** (*string*) – File name, with a default extension of *wav*.
>
> **Returns**
>
> **Return type** None

**play**()

Play audio buffer via audio jack.

Since both channels are sampled, the buffer size has to be twice the sample length.

> **Returns**
>
> **Return type** None

**record**(*seconds*)

Record data from audio controller to audio buffer.

The sample rate for both channels is 48000Hz. Note that the *sample_len* will only be changed when the buffer is modified. Since both channels are sampled, the buffer size has to be twice the sample length.

> **Parameters** **seconds** (*float*) – The number of seconds to be recorded.
>
> **Returns**
>
> **Return type** None

**save**(*file*)

Save audio buffer content to a file.

The recorded file is of format *.wav*. Note that only 24 bits out of each 32-bit sample are the real samples; the highest 8 bits are padding, which should be removed when writing the wave file.

---

**Note:** The saved file will be put into the specified path, or in the working directory in case the path does not exist.

---

> **Parameters** **file** (*string*) – File name, with a default extension of *wav*.
>
> **Returns**
>
> **Return type** None

**select_line_in**()

Select LINE_IN on the board.

This method will select the LINE_IN as the input.

**select_microphone**()

Select MIC on the board.

This method will select the MIC as the input.

**class** pynq.lib.audio.**AudioDirect**(*description*, *gpio_name=None*)
Bases: *pynq.overlay.DefaultIP*

Class to interact with audio controller.

Each audio sample is a 32-bit integer. The audio controller supports only mono mode, and uses pulse density modulation (PDM).

**mmio**
The MMIO object associated with the audio controller.

> **Type** *MMIO*

**gpio**
The GPIO object associated with the audio controller.

> **Type** *GPIO*

**buffer**
The numpy array to store the audio.

> **Type** numpy.ndarray

**sample_rate**
Sample rate of the current buffer content.

> **Type** int

**sample_len**
Sample length of the current buffer content.

> **Type** int

**bindto = ['xilinx.com:user:audio_direct:1.1']**

**bypass_start**()
Stream audio controller input directly to output.

> **Returns**

> **Return type** None

**bypass_stop**()
Stop streaming input to output directly.

> **Returns**

> **Return type** None

**static info**(*file*)
Prints information about the sound files.

The information includes name, channels, samples, frames, etc.

---

**Note:** The file will be searched in the specified path, or in the working directory in case the path does not exist.

---

> **Parameters** **file** (*string*) – File name, with a default extension of *pdm*.

> **Returns**

> **Return type** None

**load**(*file*)
>   Loads file into internal audio buffer.
>
>   The recorded file is of format *\*.pdm*.
>
>   ---
>
>   **Note:** The file will be searched in the specified path, or in the working directory in case the path does not exist.
>
>   ---
>
>   > **Parameters** **file** (`string`) – File name, with a default extension of *pdm*.
>   >
>   > **Returns**
>   >
>   > **Return type**  None

**play**()
>   Play audio buffer via audio jack.
>
>   > **Returns**
>   >
>   > **Return type**  None

**record**(*seconds*)
>   Record data from audio controller to audio buffer.
>
>   The sample rate per word is 192000Hz.
>
>   > **Parameters** **seconds** (`float`) – The number of seconds to be recorded.
>   >
>   > **Returns**
>   >
>   > **Return type**  None

**save**(*file*)
>   Save audio buffer content to a file.
>
>   The recorded file is of format *\*.pdm*.
>
>   ---
>
>   **Note:** The saved file will be put into the specified path, or in the working directory in case the path does not exist.
>
>   ---
>
>   > **Parameters** **file** (`string`) – File name, with a default extension of *pdm*.
>   >
>   > **Returns**
>   >
>   > **Return type**  None

## pynq.lib.axigpio Module

The pynq.lib.axigpio module is a driver for interacting with the Xilinx AXIGPIO IP Block. Each AXI GPIO IP instantiated in the fabric has at least one, and at most two channels.

**class** pynq.lib.axigpio.**AxiGPIO**(*description*)
>   Bases: *pynq.overlay.DefaultIP*
>
>   Class for interacting with the AXI GPIO IP block.
>
>   This class exposes the two banks of GPIO as the *channel1* and *channel2* attributes. Each channel can have the direction and the number of wires specified.

The wires in the channel can be accessed from the channel using slice notation - all slices must have a stride of 1. Input wires can be *read* and output wires can be written to, toggled, or turned off or on. InOut channels combine the functionality of input and output channels. The tristate of the pin is determined by whether the pin was last read or written.

**class Channel**(*parent*, *channel*)
> Bases: `object`

> Class representing a single channel of the GPIO controller.

> Wires are and bundles of wires can be accessed using array notation with the methods on the wires determined by the type of the channel:

```
input_channel[0].read()
output_channel[1:3].on()
```

> This class instantiated not used directly, instead accessed through the *AxiGPIO* classes attributes. This class exposes the wires connected to the channel as an array or elements. Slices of the array can be assigned simultaneously.

> **read**()
>> Read the state of the input pins

> **setdirection**(*direction*)
>> Set the direction of the channel

>> Must be one of AxiGPIO.{Input, Output, InOut} or the string 'in', 'out', or 'inout'

> **setlength**(*length*)
>> Set the number of wires connected to the channel

> **trimask**
>> Gets or sets the tri-state mask for an inout channel

> **wait_for_interrupt_async**()
>> Wait for the interrupt on the channel to be signalled

>> This is intended to be used by slices waiting for a particular value but can be used in any situation to wait for a per-channel interrupt.

> **write**(*val*, *mask*)
>> Set the state of the output pins

**class InOut**(*parent*, *start*, *stop*)
> Bases: `pynq.lib.axigpio.Output`, `pynq.lib.axigpio.Input`

> Class representing wires in an inout channel.

> This class should be passed to *setdirection* to indicate the channel should be used for both input and output. It should not be used directly.

> **read**()
>> Reads the value of all the wires in the slice

>> Changes the tristate of the slice to input. If there is more than one wire in the slice then the least significant bit of the return value corresponds to the wire with the lowest index.

> **write**(*val*)
>> Set the value of the slice

>> Changes the tristate of the slice to output. If the slice consists of more than one wire then the least significant bit of *val* corresponds to the lowest index wire.

**class Input**(*parent*, *start*, *stop*)
    Bases: `object`

    Class representing wires in an input channel.

    This class should be passed to *setdirection* to indicate the channel should be used for input only. It should not be used directly.

    **read**()
        Reads the value of all the wires in the slice

        If there is more than one wire in the slice then the least significant bit of the return value corresponds to the wire with the lowest index.

    **wait_for_value**(*value*)
        Wait until the specified value is read

        This function is dependent on interrupts being enabled and will throw a *RuntimeError* otherwise. Internally it uses asyncio so should not be used inside an asyncio task. Use *wait_for_value_async* if using asyncio.

    **wait_for_value_async**(*value*)
        Coroutine that waits until the specified value is read

        This function relies on interrupts being available for the IP block and will throw a *RuntimeError* otherwise.

**class Output**(*parent*, *start*, *stop*)
    Bases: `object`

    Class representing wires in an output channel.

    This class should be passed to *setdirection* to indicate the channel should be used for output only. It should not be used directly.

    **off**()
        Turns off all of the wires in the slice

    **on**()
        Turns on all of the wires in the slice

    **read**()
        Reads the value of all the wires in the slice

        If there is more than one wire in the slice then the least significant bit of the return value corresponds to the wire with the lowest index.

    **toggle**()
        Toggles all of the wires in the slice

    **write**(*val*)
        Set the value of the slice

        If the slice consists of more than one wire then the least significant bit of *val* corresponds to the lowest index wire.

**bindto = ['xilinx.com:ip:axi_gpio:2.0']**

**setdirection**(*direction*, *channel=1*)
    Sets the direction of a channel in the controller

    Must be one of AxiGPIO.{Input, Output, InOut} or the string 'in', 'out' or 'inout'

**setlength**(*length*, *channel=1*)
    Sets the length of a channel in the controller

### pynq.lib.iic Module

The pynq.lib.iic module is a driver for interacting with the Xilinx Axi IIC IP Block.

**class** `pynq.lib.iic.`**`AxiIIC`**(*description*)

> Bases: *`pynq.overlay.DefaultIP`*
>
> Driver for the AXI IIC controller
>
> **`REPEAT_START = 1`**
>
> **`bindto = ['xilinx.com:ip:axi_iic:2.0']`**
>
> **`receive`**(*address*, *data*, *length*, *option=0*)
>
>> Receive data from an attached IIC slave
>>
>>> **Parameters**
>>>
>>> - **`address`** (*int*) – Address of the slave device
>>> - **`data`** (*bytes-like*) – Data to receive
>>> - **`length`** (*int*) – Number of bytes to receive
>>> - **`option`** (*int*) – Optionally *REPEAT_START* to keep hold of the bus between transactions
>
> **`send`**(*address*, *data*, *length*, *option=0*)
>
>> Send data to an attached IIC slave
>>
>>> **Parameters**
>>>
>>> - **`address`** (*int*) – Address of the slave device
>>> - **`data`** (*bytes-like*) – Data to send
>>> - **`length`** (*int*) – Length of data
>>> - **`option`** (*int*) – Optionally *REPEAT_START* to keep hold of the bus between transactions
>
> **`wait`**()
>
>> Wait for the transaction to complete

### pynq.lib.button Module

The pynq.lib.rgbled module is a driver for reading values from onboard push-buttons and waiting for button-triggered events.

**class** `pynq.lib.button.`**`Button`**(*device*)

> Bases: `object`
>
> This class controls the onboard push-buttons.
>
> **`_impl`**
>
>> An object with appropriate Button methods
>>
>>> **Type** object
>
> **`read`**()
>
>> Read the current value of the button.
>
> **`wait_for_value`**(*value*)
>
>> Wait for the button to be pressed or released.

> Parameters **value** (*int*) – 1 to wait for press or 0 to wait for release

## pynq.lib.dma Module

**class** `pynq.lib.dma.`**DMA**(*description*, *\*args*, *\*\*kwargs*)

> Bases: *pynq.overlay.DefaultIP*

Class for Interacting with the AXI Simple DMA Engine

This class provides two attributes for the read and write channels. The read channel copies data from the stream into memory and the write channel copies data from memory to the output stream. Both channels have an identical API consisting of *transfer* and *wait* functions. If interrupts have been enabled and connected for the DMA engine then *wait_async* is also present.

Buffers to be transferred must be a *PynqBuffer* objectedallocated through *pynq.allocate()* function either directly or indirectly. This means that Frames from the video subsystem can be transferred using this class.

**recvchannel**

> The stream to memory channel
>
> > **Type** _DMAChannel

**sendchannel**

> The memory to stream channel
>
> > **Type** _DMAChannel

**bindto = ['xilinx.com:ip:axi_dma:7.1']**

## pynq.lib.led Module

The pynq.lib.rgbled module is a driver for controlling onboard single-color Light Emitting Diodes (LEDs).

**class** `pynq.lib.led.`**LED**(*device*)

> Bases: `object`

This class controls the onboard leds.

**_impl**

> An object with appropriate LED methods
>
> > **Type** object

**off**()

> Turn off led.

**on**()

> Turn on led.

**toggle**()

> Toggle led on/off.

## pynq.lib.logictools Package

## pynq.lib.logictools.boolean_generator Module

**class** `pynq.lib.logictools.boolean_generator.`**BooleanGenerator**(*mb_info*,
*intf_spec_name='PYNQZ1_LOGICTOOLS_*

> Bases: `object`

Class for the Boolean generator.

This class can implement any combinational function on user IO pins. Since each LUT5 takes 5 inputs, the basic function that users can implement is 5-input, 1-output boolean function. However, by concatenating multiple LUT5 together, users can implement complex boolean functions.

There are 20 5-LUTs, so users can implement at most 20 basic boolean functions at a specific time.

**logictools_controller**
> The generator controller for this class.

> > **Type** LogicToolsController

**intf_spec**
> The interface specification, e.g., PYNQZ1_LOGICTOOLS_SPECIFICATION.

> > **Type** dict

**expressions**
> The boolean expressions, each expression being a string.

> > **Type** list/dict

**waveforms**
> A dictionary storing the waveform objects for display purpose.

> > **Type** dict

**input_pins**
> A list of input pins used by the generator.

> > **Type** list

**output_pins**
> A list of output pins used by the generator.

> > **Type** list

**analyzer**
> Analyzer to analyze the raw capture from the pins.

> > **Type** *TraceAnalyzer*

**num_analyzer_samples**
> Number of analyzer samples to capture.

> > **Type** int

**frequency_mhz**
> The frequency of the captured samples, in MHz.

> > **Type** float

**analyze()**
> Update the captured samples.

> This method updates the captured samples from the trace analyzer. It is required after each step() / run()

**clear_wave()**
> Clear the waveform object so new patterns can be accepted.

> This function is required after each *stop()*.

**connect()**
> Method to configure the IO switch.

> Usually this method should only be used internally. Users only need to use *run()* method.

**disconnect**()
> Method to disconnect the IO switch.

> Usually this method should only be used internally. Users only need to use *stop()* method.

**reset**()
> Reset the boolean generator.

> This method will bring the generator from any state to 'RESET' state.

**run**()
> Run the boolean generator.

> The method will first collects the pins used and sends the list to Microblaze for handling. Then it will start to run the boolean generator.

**setup**(*expressions*, *frequency_mhz=10*)
> Configure the generator with new boolean expression.

> This method will bring the generator from 'RESET' to 'READY' state.

>> **Parameters**

>>> • **expressions** (*list/dict*) – The boolean expression to be configured.

>>> • **frequency_mhz** (*float*) – The frequency of the captured samples, in MHz.

**show_waveform**()
> Display the boolean logic generator in a Jupyter notebook.

> A wavedrom waveform is shown with all inputs and outputs displayed.

**status**
> Return the generator's status.

>> **Returns** Indicating the current status of the generator; can be 'RESET', 'READY', or 'RUN-NING'.

>> **Return type** str

**step**()
> Step the boolean generator.

> The method will first collects the pins used and sends the list to Microblaze for handling. Then it will start to step the boolean generator.

**stop**()
> Stop the boolean generator.

> This method will stop the currently running boolean generator.

**trace**(*use_analyzer=True*, *num_analyzer_samples=128*)
> Configure the trace analyzer.

> By default, the trace analyzer is always on, unless users explicitly disable it.

>> **Parameters**

>>> • **use_analyzer** (*bool*) – Whether to use the analyzer to capture the trace.

>>> • **num_analyzer_samples** (*int*) – The number of analyzer samples to capture.

### pynq.lib.logictools.fsm_generator Module

**class** pynq.lib.logictools.fsm_generator.**FSMGenerator**(*mb_info*,

*intf_spec_name='PYNQZ1_LOGICTOOLS_SPECIFIC*

Bases: `object`

Class for Finite State Machine generator.

This class enables users to specify a Finite State Machine (FSM). Users have to provide a FSM in the following format.

fsm_spec = {'inputs': [('reset','D0'), ('direction','D1')],

'outputs': [('alpha','D3'), ('beta','D4'), ('gamma','D5')],

'states': ('S0', 'S1', 'S2', 'S3', 'S4', 'S5'),

'transitions': [['00', 'S0', 'S1', '000'],

>   ['01', 'S0', 'S5', '000'],
>
>   ['00', 'S1', 'S2', '001'],
>
>   ['01', 'S1', 'S0', '001'],
>
>   ['00', 'S2', 'S3', '010'],
>
>   ['01', 'S2', 'S1', '010'],
>
>   ['00', 'S3', 'S4', '011'],
>
>   ['01', 'S3', 'S2', '011'],
>
>   ['00', 'S4', 'S5', '100'],
>
>   ['01', 'S4', 'S3', '100'],
>
>   ['00', 'S5', 'S0', '101'],
>
>   ['01', 'S5', 'S4', '101'],
>
>   ['1-', '*', 'S0', '']]}

The current implementation assumes Moore machine, so the output is decided by the current state. Hence, if a wildcard * is specified for the current state, users can just set the output to be empty.

**logictools_controller**
>   The generator controller for this class.
>
>   >   **Type** LogicToolsController

**intf_spec**
>   The interface specification, e.g., PYNQZ1_LOGICTOOLS_SPECIFICATION.
>
>   >   **Type** dict

**fsm_spec**
>   The FSM specification, with inputs (list), outputs (list), states (list), and transitions (list).
>
>   >   **Type** dict

**num_input_bits**
>   The number of input bits / pins.
>
>   >   **Type** int

**num_outputs**
>   The number of possible FSM outputs specified by users.

> **Type** int

**num_output_bits**
> The number of bits used for the FSM outputs.
>
> > **Type** int

**num_states**
> The number of FSM states specified by users.
>
> > **Type** int

**num_state_bits**
> The number of bits used for the FSM states.
>
> > **Type** int

**state_names**
> List of state names specified by the users.
>
> > **Type** list

**transitions**
> Transition list with all the wildcards replaced properly.
>
> > **Type** int

**input_pins**
> List of input pins on Arduino header.
>
> > **Type** list

**output_pins**
> List of output pins on Arduino header.
>
> > **Type** list

**use_state_bits**
> Flag indicating whether the state bits are shown on output pins.
>
> > **Type** bool

**analyzer**
> Analyzer to analyze the raw capture from the pins.
>
> > **Type** *TraceAnalyzer*

**num_analyzer_samples**
> The number of analyzer samples to capture.
>
> > **Type** int

**frequency_mhz**
> The frequency of the running FSM / captured samples, in MHz.
>
> > **Type** float

**waveform**
> The Waveform object used for Wavedrom display.
>
> > **Type** *Waveform*

**analyze()**
> Update the captured samples.
>
> This method updates the captured samples from the trace analyzer. It is required after each step() / run().

---

**clear_wave**()
> Clear the waveform object so new patterns can be accepted.

> This function is required after each *stop()*.

**connect**()
> Method to configure the IO switch.

> Usually this method should only be used internally. Users only need to use *run()* method.

**disconnect**()
> Method to disconnect the IO switch.

> Usually this method should only be used internally. Users only need to use *stop()* method.

**reset**()
> Reset the FSM generator.

> This method will bring the generator from any state to 'RESET' state.

**run**()
> Run the FSM generator.

> The method will first collects the pins used and sends the list to Microblaze for handling. Then it will start to run the FSM generator.

**setup** (*fsm_spec*, *use_state_bits=False*, *frequency_mhz=10*)
> Configure the programmable FSM generator.

> This method will configure the FSM based on supplied configuration specification. Users can send the samples to PatternAnalyzer for additional analysis.

> > **Parameters**

> > - **fsm_spec** (`dict`) – The FSM specification, with inputs (list), outputs (list), states (list), and transitions (list).

> > - **use_state_bits** (`bool`) – Whether to check the state bits in the final output pins.

> > - **frequency_mhz** (`float`) – The frequency of the FSM and captured samples, in MHz.

**show_state_diagram** (*file_name='fsm_spec.png'*, *save_png=False*)
> Display the state machine in Jupyter notebook.

> This method uses the installed package *pygraphviz*. References: [http://pygraphviz.github.io/documentation/latest/pygraphviz.pdf](http://pygraphviz.github.io/documentation/latest/pygraphviz.pdf)

> A PNG file of the state machine will also be saved into the current working directory.

> > **Parameters**

> > - **file_name** (`str`) – The name / path of the picture for the FSM diagram.

> > - **save_png** (`bool`) – Whether to save the PNG showing the state diagram.

**show_waveform**()
> Display the waveform.

> This method requires the waveform class to be present. Also, javascripts will be copied into the current directory.

**status**
> Return the generator's status.

> > **Returns** Indicating the current status of the generator; can be 'RESET', 'READY', or 'RUNNING'.

> **Return type** str

**step**()
> Step the FSM generator.

> The method will first collects the pins used and sends the list to Microblaze for handling. Then it will start to step the FSM generator.

**stop**()
> Stop the FSM generator.

> This command will stop the pattern generated from FSM.

**trace**(*use_analyzer=True*, *num_analyzer_samples=128*)
> Configure the trace analyzer.

> By default, the trace analyzer is always on, unless users explicitly disable it.

> **Parameters**

>> • **use_analyzer** (`bool`) – Whether to use the analyzer to capture the trace.

>> • **num_analyzer_samples** (`int`) – The number of analyzer samples to capture.

pynq.lib.logictools.fsm_generator.**check_duplicate**(*fsm_spec*, *key*)
> Function to check duplicate entries in a nested dictionary.

> This method will check the entry indexed by key in a dictionary. An exception will be raised if there are duplicated entries.

> **Parameters**

>> • **fsm_spec** (`dict`) – The dictionary where the check to be made.

>> • **key** (`object`) – The key to index the dictionary.

pynq.lib.logictools.fsm_generator.**check_moore**(*num_states*, *num_outputs*)
> Check whether the specified state machine is a moore machine.

> This method will raise an exception if there are more state outputs than the number of states.

> **Parameters**

>> • **num_states** (`int`) – The number of bits used for states.

>> • **num_outputs** (`int`) – The number of state outputs.

pynq.lib.logictools.fsm_generator.**check_num_bits**(*num_bits*, *label*, *minimum=0*, *maximum=32*)
> Check whether the number of bits are still in a valid range.

> This method will raise an exception if *num_bits* is out of range.

> **Parameters**

>> • **num_bits** (`int`) – The number of bits of a specific field.

>> • **label** (`str`) – The label of the field.

>> • **minimum** (`int`) – The minimum number of bits allowed in that field.

>> • **maximum** (`int`) – The maximum number of bits allowed in that field.

pynq.lib.logictools.fsm_generator.**check_pin_conflict**(*pins1*, *pins2*)
> Function to check whether there is conflict between input / output pins.

> This method will raise an exception if there are pins specified in both inputs and outputs.

> **Parameters**

- **pins1** (*list*) – The list of the first set of pins.

- **pins2** (*list*) – The list of the second set of pins.

pynq.lib.logictools.fsm_generator.**check_pins**(*fsm_spec*, *key*, *intf_spec*)

Check whether the pins specified are in a valid range.

This method will raise an exception if *pin* is out of range.

> **Parameters**
>
> - **fsm_spec** (*dict*) – The dictionary where the check to be made.
>
> - **key** (*object*) – The key to index the dictionary.
>
> - **intf_spec** (*dict*) – An interface spec containing the pin map.

pynq.lib.logictools.fsm_generator.**expand_transition**(*transition*, *input_list*)

Add new (partially) expanded state transition.

> **Parameters**
>
> - **transition** (*list*) – Specifies a state transition.
>
> - **input_list** (*list*) – List of inputs, where each input is a string.
>
> **Returns** New (partially) expanded state transition.
>
> **Return type** list

pynq.lib.logictools.fsm_generator.**get_bram_addr_offsets**(*num_states*, *num_input_bits*)

Get address offsets from given number of states and inputs.

This method returns the index offset for input bits. For example, if less than 32 states are used, then the index offset will be 5. If the number of states used is greater than 32 but less than 64, then the index offset will be 6.

This method also returns the address offsets for BRAM data. The returned list contains 2**'num_input_bits' offsets. The distance between 2 address offsets is 2**'index_offset'.

> **Parameters**
>
> - **num_states** (*int*) – The number of states in the state machine.
>
> - **num_input_bits** (*int*) – The number of inputs in the state machine.
>
> **Returns** A list of 2**'num_input_bits' offsets.
>
> **Return type** int, list

pynq.lib.logictools.fsm_generator.**merge_to_length**(*a*, *b*, *length*)

Merge 2 lists into a specific length.

This method will merge 2 lists into a short list, replacing the last few items of the first list if necessary.

For example, a = [1,2,3], b = [4,5,6,7], and length = 6. The result will be [1,2,4,5,6,7]. If length = 5, the result will be [1,4,5,6,7]. If length is greater or equal to 7, the result will be [1,2,3,4,5,6,7].

> **Parameters**
>
> - **a** (*list*) – A list of elements.
>
> - **b** (*list*) – Another list of elements.
>
> - **length** (*int*) – The length of the result list.
>
> **Returns** A merged list of the specified length.
>
> **Return type** list

`pynq.lib.logictools.fsm_generator.`**`replace_wildcard`**(*input_list*)
    Method to replace a wildcard - in the input values.

    This method will replace the wildcard - in the input list; the returned two lists have different values on the position of -.

    Example: ['0', '-', '1'] => (['0', '0', '1'], ['0', '1', '1'])

    >    **Parameters input_list** (*list*) – A list of multiple values, possibly with - inside.

    >    **Returns** Two lists differ by the location of -.

    >    **Return type** list,list

## pynq.lib.logictools.pattern_generator Module

**class** `pynq.lib.logictools.pattern_generator.`**`PatternGenerator`**(*mb_info*,
                                                                            *intf_spec_name='PYNQZ1_LOGICTOOLS_*
    Bases: `object`

    Class for the Pattern generator.

    This class can generate digital IO patterns / stimulus on output pins. Users can specify whether to use a pin as input or output.

    **`logictools_controller`**
        The generator controller for this class.

            **Type** LogicToolsController

    **`intf_spec`**
        The interface specification, e.g., PYNQZ1_LOGICTOOLS_SPECIFICATION.

            **Type** dict

    **`stimulus_group`**
        A group of stimulus wavelanes.

            **Type** dict

    **`stimulus_group_name`**
        The name of the stimulus wavelanes.

            **Type** str

    **`stimulus_names`**
        The list of all the stimulus wavelane names, each name being a string.

            **Type** list

    **`stimulus_pins`**
        The list of all the stimulus wavelane pin labels, each pin label being a string.

            **Type** list

    **`stimulus_waves`**
        The list of all the stimulus wavelane waves, each wave being a string consisting of wavelane tokens.

            **Type** list

    **`analysis_group`**
        A group of analysis wavelanes.

            **Type** dict

**analysis_group_name**
> The name of the analysis wavelanes.

>> **Type** str

**analysis_names**
> The list of all the analysis wavelane names, each name being a string.

>> **Type** list

**analysis_pins**
> The list of all the analysis wavelane pin labels, each pin label being a string.

>> **Type** list

**src_samples**
> The numpy array storing the stimuli, each sample being 32 bits.

>> **Type** numpy.ndarray

**dst_samples**
> The numpy array storing the response, each sample being 64 bits.

>> **Type** numpy.ndarray

**waveform_dict**
> A dictionary storing the patterns in WaveJason format.

>> **Type** dict

**waveform**
> The Waveform object used for Wavedrom display.

>> **Type** *Waveform*

**analyzer**
> Analyzer to analyze the raw capture from the pins.

>> **Type** *TraceAnalyzer*

**num_analyzer_samples**
> The number of analyzer samples to capture.

>> **Type** int

**frequency_mhz**
> The frequency of the running generator / captured samples, in MHz.

>> **Type** float

**analyze()**
> Update the captured samples.

> This method updates the captured samples from the trace analyzer. It is required after each step() / run()

**clear_wave()**
> Clear the waveform object so new patterns can be accepted.

> This function is required after each *stop()*.

**connect()**
> Method to configure the IO switch.

> Usually this method should only be used internally. Users only need to use *run()* method.

**disconnect**()
> Method to disconnect the IO switch.

> Usually this method should only be used internally. Users only need to use *stop()* method.

**longest_wave**
> Return the name of the longest wave.

> Will only be changed by internal method.

**max_wave_length**
> Return the maximum wave length

> Will only be changed by internal method.

**reset**()
> Reset the pattern generator.

> This method will bring the generator from any state to 'RESET' state.

**run**()
> Run the pattern generation.

> The method will first collects the pins used and sends the list to Microblaze for handling. Then it will start to run the pattern generator.

**setup**(*waveform_dict*, *stimulus_group_name=None*, *analysis_group_name=None*, *mode='single'*, *frequency_mhz=10*)
> Configure the pattern generator with a single bit pattern.

> Generates a bit pattern for a single shot operation at specified IO pins with the specified number of samples.

> Each bit of the 20-bit patterns, from LSB to MSB, corresponds to: D0, D1, . . . , D13, A0, A1, . . . , A5, respectively.

> Note the all the lanes should have the same number of samples. And the token inside wave are already converted into bit string.

> Users can ignore the returned data in case only the pattern generator is required.

> Mode *single* means the pattern will be generated only once, while mode *multiple* means the pattern will be generated repeatedly.

> > **Parameters**
> >
> > - **waveform_dict** (`dict`) – Waveform dictionary in WaveJSON format.
> >
> > - **stimulus_group_name** (`str`) – Name of the WaveLane group for the stimulus if used.
> >
> > - **analysis_group_name** (`str`) – Name of the WaveLane group for the analysis if used.
> >
> > - **mode** (`str`) – Mode of the pattern generator, can be *single* or *multiple*.
> >
> > - **frequency_mhz** (`float`) – The frequency of the captured samples, in MHz.

**show_waveform**()
> Display the waveform in Jupyter notebook.

> This method requires the waveform class to be present. At the same time, javascripts will be copied into the current directory.

**status**
> Return the generator's status.

> **Returns** Indicating the current status of the generator; can be 'RESET', 'READY', or 'RUN-NING'.
>
> **Return type** str

**step**()

> Step the pattern generator.
>
> The method will first collects the pins used and sends the list to Microblaze for handling. Then it will start to step the pattern generator.

**stop**()

> Stop the pattern generation.
>
> This method will stop the currently running pattern generator.

**trace**(*use_analyzer=True*, *num_analyzer_samples=128*)

> Configure the trace analyzer.
>
> By default, the trace analyzer is always on, unless users explicitly disable it.
>
> > **Parameters**
> >
> > - **use_analyzer** (*bool*) – Whether to use the analyzer to capture the trace.
> >
> > - **num_analyzer_samples** (*int*) – The number of analyzer samples to capture.

### pynq.lib.logictools.trace_analyzer Module

**class** pynq.lib.logictools.trace_analyzer.**TraceAnalyzer**(*ip_info*,
                                           *intf_spec_name='PYNQZ1_LOGICTOOLS_SPECIF*

> Bases: object
>
> Class for trace analyzer.
>
> This class can capture digital IO patterns / stimulus on monitored pins.
>
> This class can wrap one out of the two classes: (1) the Microblaze controlled trace analyzer, or (2) the PS controlled trace analyzer.
>
> To use the PS controlled trace analyzer, users can set the *ip_info* to a dictionary containing the corresponding IP name; for example:

```
>>> ip_info = {'trace_cntrl':'trace_analyzer_pmoda/trace_cntrl_0',
       'trace_dma': 'trace_analyzer_pmoda/axi_dma_0'}
```

> Otherwise the Microblaze controlled trace analyzer will be used. By default, the Microblaze controlled version will be used, and the interface specification name will be set to *PYNQZ1_LOGICTOOLS_SPECIFICATION*.
>
> Most of the methods implemented inside this class assume the protocol is known, so the pattern can be decoded and added to the annotation of the waveforms.
>
> In case the protocol is unknown, users should refrain from using these methods.
>
> Two files are maintained by this class: the *csv* file, which is human readable; and the *sr* file, which is sigrok readable.
>
> **analyze**(*steps=0*)
>
> > Analyze the captured pattern.
> >
> > This function will process the captured pattern and put the pattern into a Wavedrom compatible format.
> >
> > The data output is of format:

[{'name': '', 'pin': 'D1', 'wave': '1...0.....'}, {'name': '', 'pin': 'D2', 'wave': '0.1..01.01'}]

Note the all the lanes should have the same number of samples. All the pins are assumed to be tri-stated and traceable.

Currently only no *step()* method is supported for PS controlled trace analyzer.

> **Parameters steps** (*int*) – Number of samples to analyze. A value 0 means to analyze all the valid samples.
>
> **Returns** A list of dictionaries, each dictionary consisting the pin number, and the waveform pattern in string format.
>
> **Return type** list

**decode**(*trace_csv*, *start_pos*, *stop_pos*, *decoded_file*, *options=''*)
> Parse CSV file, add metadata, and use sigrok to decode transactions.
>
> Internally, this method is calling *save_csv()*, *set_metadata()*, and *sigrok_decode()* methods.
>
> **Parameters**
>
> - **trace_csv** (*str*) – Name of the output file (*\*.csv*) which can be opened in text editor.
> - **start_pos** (*int*) – Starting sample number, no less than 1.
> - **stop_pos** (*int*) – Stopping sample number, no more than the maximum number of samples.
> - **decoded_file** (*str*) – Name of the output file, which can be opened in text editor.
> - **options** (*str*) – Additional options to be passed to sigrok-cli.
>
> **Returns**
>
> **Return type** None

**get_transactions**()
> List all the transactions captured.
>
> The transaction list will only be non-empty after users have run *decode()* method. An exception will be raised if the transaction is empty, or the text file cannot be found.
>
> **Returns** A list of dictionaries. Each bus event is a dictionary: [{'command': str, 'begin': int, 'end': int}]
>
> **Return type** list

**reset**()
> Reset the trace analyzer.
>
> This method will bring the trace analyzer from any state to 'RESET' state.
>
> At the same time, all the trace files stored previously will be removed.

**run**()
> Start the trace capture.
>
> **Returns**
>
> **Return type** None

**set_protocol**(*protocol*, *probes*)
> Set the protocol and probes for the decoder.
>
> This method is usually called at beginning of the analyzer. To learn from that specific protocol, users can call *show_protocol* to learn useful information about that protocol.

Currently only *i2c* and *spi* are supported.

This method also sets the probe names for the decoder.

The dictionary *probes* depends on the protocol. For instance, the I2C protocol requires the keys 'SCL' and 'SDA'. An example can be:

>>>probes = {'SCL': 'D2', 'SDA': 'D3'}

To avoid memory error for decoding, users can add *NC* as non-used pins to the probes.

> **Parameters**
>
> - **protocol** (`str`) – The name of the protocol.
>
> - **probes** (`dict`) – A dictionary keeping the probe names and pin number.

**setup**(*num_analyzer_samples=128*, *frequency_mhz=10*, *fclk_index=None*)
  Configure the trace analyzer.

  The wrapper method for configuring the PS or Microblaze controlled trace analyzer.

  Users need to provide the *fclk_index* explicitly, otherwise the driver will just use the default clock. For MB-controlled trace analyzer, the default *fclk_index* is 1; for PS-controlled trace analyzer, the default *fclk_index* is 3.

  > **Parameters**
  >
  > - **num_analyzer_samples** (`int`) – The number of samples to be analyzed.
  >
  > - **frequency_mhz** (`float`) – The frequency of the captured samples, in MHz.
  >
  > - **fclk_index** (`int`) – The index of the fclk controlled by clock management object.

**show_protocol**()
  Show information about the specified protocol.

  This method will print out useful information about the protocol.

  > **Returns**
  >
  > **Return type** None

**status**
  Return the analyzer's status.

  > **Returns** Indicating the current status of the analyzer; can be 'RESET', 'READY', or 'RUNNING'.
  >
  > **Return type** str

**step**()
  Step the trace analyzer.

  This method is only supported in the Microblaze controlled trace analyzer. An exception will be raised if users want to call this method in PS controlled trace analyzer.

**stop**()
  Stop the DMA after capture is done.

  > **Returns**
  >
  > **Return type** None

pynq.lib.logictools.trace_analyzer.**get_tri_state_pins**(*io_pin_dict*, *tri_dict*)
  Function to check tri-state pin specifications.

Any tri-state pin requires the input/output pin, and the tri-state selection pin to be specified. If any one is missing, this method will raise an exception.

> **Parameters**
>
> > - **io_pin_dict** (*dict*) – A dictionary storing the input/output pin mapping.
> > - **tri_dict** (*dict*) – A dictionary storing the tri-state pin mapping.
>
> **Returns** A list storing unique tri-state and non tri-state pin names.
>
> **Return type** list

## pynq.lib.logictools.waveform Module

**class** pynq.lib.logictools.waveform.**Waveform**(*waveform_dict*, *intf_spec_name='PYNQZ1_LOGICTOOLS_SPECIFICATION'*, *stimulus_group_name=None*, *analysis_group_name=None*)

Bases: `object`

A wrapper class for Wavedrom package and interfacing functions.

This class wraps the key functions of the Wavedrom package, including waveform display, bit pattern converting, csv converting, etc.

A typical example of the waveform dictionary is:

```
>>> loopback_test = {'signal': [
```

> ['stimulus',
>
> {'name': 'clk0', 'pin': 'D0', 'wave': 'lh' * 64},
>
> {'name': 'clk1', 'pin': 'D1', 'wave': 'l.h.' * 32},
>
> {'name': 'clk2', 'pin': 'D2', 'wave': 'l...h...' * 16},
>
> {'name': 'clk3', 'pin': 'D3', 'wave': 'l......h......' * 8}],
>
> ['analysis',
>
> {'name': 'clk15', 'pin': 'D15'},
>
> {'name': 'clk16', 'pin': 'D16'},
>
> {'name': 'clk17', 'pin': 'D17'},
>
> {'name': 'clk18', 'pin': 'D18'},
>
> {'name': 'clk19', 'pin': 'D19'}]

],

'foot': {'tock': 1},

'head': {'text': 'Loopback Test'}}

**waveform_dict**
     The json data stored in the dictionary.

> **Type** dict

**intf_spec**

   The interface specification, e.g., PYNQZ1_LOGICTOOLS_SPECIFICATION.

   > **Type** dict

**stimulus_group_name**

   Name of the WaveLane group for the stimulus, defaulted to *stimulus*.

   > **Type** str

**analysis_group_name**

   Name of the WaveLane group for the analysis, defaulted to *analysis*.

   > **Type** str

**stimulus_group**

   A group of lanes, each lane being a dict of name, pin label,and wave.

   > **Type** list

**analysis_group**

   A group of lanes, each lane being a dict of name, pin label,and wave.

   > **Type** list

**analysis_group**

   Return the analysis WaveLane group.

   An analysis group looks like: [{'name': 'dat', 'pin': 'D1', 'wave': 'l…h…lhlh'}, {'name': 'req', 'pin': 'D2', 'wave': 'lhlhlhlh.…'}]

   > **Returns** A list of lanes, each lane being a dictionary of name, pin label, and wave.

   > **Return type** list

**analysis_names**

   Returns all the names of the analysis WaveLanes.

   The returned names are in the same order as in the waveform dictionary.

   > **Returns** A list of names for all the analysis WaveLanes.

   > **Return type** list

**analysis_pins**

   Returns all the pin labels of the analysis WaveLanes.

   The returned pin labels are in the same order as in the waveform dictionary.

   > **Returns** A list of pin labels for all the analysis WaveLanes.

   > **Return type** list

**analysis_waves**

   Returns all the waves of the analysis WaveLanes.

   The returned waves are in the same order as in the waveform dictionary.

   > **Returns** A list of waves for all the analysis WaveLanes.

   > **Return type** list

**annotate**(*group_name*, *wavelane_group*)

   Add annotation to the existing waveform dictionary.

   This method will add annotation wavelane into the specified group. Usually this is used in combination with the trace analyzer.

---

The annotation usually has the following format:

[{**name: '',** wave: 'x.444x4.x', data: ['read', 'write', 'read', 'data']}]

> **Parameters**
>
> • **group_name** (*str*) – The name of the WaveLane group to be updated.
>
> • **wavelane_group** (*list*) – The WaveLane group specified for updating.

**append**(*group_name*, *wavelane_group*)

Append new data to the existing waveform dictionary.

A typical use case of this method is that it gets the output returned by the analyzer and append new data to the dictionary.

Since the analyzer only knows the pin labels, the data returned from the pattern analyzer is usually of format:

[{'name': '', 'pin': 'D1', 'wave': 'l...h...lhlh'}, {'name': '', 'pin': 'D2', 'wave': 'lhlhlhlh....'}]

Note the all the lanes should have the same number of samples. Note each lane in the analysis group has its pin number. Based on this information, this function only updates the lanes specified.

> **Parameters**
>
> • **group_name** (*str*) – The name of the WaveLane group to be updated.
>
> • **wavelane_group** (*list*) – The WaveLane group specified for updating.

**clear_wave**(*group_name*)

Clear the wave in the existing waveform dictionary.

This method will clear the wave stored in each wavelane, so that a brand-new waveform dict can be constructed.

Annotation is assumed to have an empty name, so the entire annotation lane will get deleted in this method.

> **Parameters group_name** (*str*) – The name of the WaveLane group to be updated.

**display**()

Display the waveform using the Wavedrom package.

This package requires 2 javascript files to be copied locally.

**stimulus_group**

Return the stimulus WaveLane group.

A stimulus group looks like: [{'name': 'dat', 'pin': 'D1', 'wave': 'l...h...lhlh'}, {'name': 'req', 'pin': 'D2', 'wave': 'lhlhlhlh....'}]

> **Returns** A list of lanes, each lane being a dictionary of name, pin label, and wave.
>
> **Return type** list

**stimulus_names**

Returns all the names of the stimulus WaveLanes.

The returned names are in the same order as in the waveform dictionary.

> **Returns** A list of names for all the stimulus WaveLanes.
>
> **Return type** list

**stimulus_pins**
> Returns all the pin labels of the stimulus WaveLanes.
>
> The returned pin labels are in the same order as in the waveform dictionary.
>
> > **Returns** A list of pin labels for all the stimulus WaveLanes.
> >
> > **Return type** list

**stimulus_waves**
> Returns all the waves of the stimulus WaveLanes.
>
> The returned waves are in the same order as in the waveform dictionary.
>
> > **Returns** A list of waves for all the stimulus WaveLanes.
> >
> > **Return type** list

**update**(*group_name*, *wavelane_group*)
> Update waveform dictionary based on the specified WaveLane group.
>
> A typical use case of this method is that it gets the output returned by the analyzer and refreshes the data stored in the dictionary.
>
> Since the analyzer only knows the pin labels, the data returned from the pattern analyzer is usually of format:
>
> [{'name': '', 'pin': 'D1', 'wave': 'l...h...lhlh'}, {'name': '', 'pin': 'D2', 'wave': 'lhlhlhlh....'}]
>
> Note the all the lanes should have the same number of samples. Note each lane in the analysis group has its pin number. Based on this information, this function only updates the lanes specified.
>
> > **Parameters**
> >
> > - **group_name** (*str*) – The name of the WaveLane group to be updated.
> >
> > - **wavelane_group** (*list*) – The WaveLane group specified for updating.

pynq.lib.logictools.waveform.**bitstring_to_int**(*bitstring*)
> Function to convert a bit string to integer list.
>
> For example, if the bit string is '0110', then the integer list will be [0,1,1,0].
>
> > **Parameters bitstring** (*str*) – The input string to convert.
> >
> > **Returns** A list of elements, each element being 0 or 1.
> >
> > **Return type** list

pynq.lib.logictools.waveform.**bitstring_to_wave**(*bitstring*)
> Function to convert a pattern consisting of *0*, *1* into a sequence of *l*, *h*, and dots.
>
> For example, if the bit string is "010011000111", then the result will be "lhl.h.l..h.".
>
> > **Returns** New wave tokens with valid tokens and dots.
> >
> > **Return type** str

pynq.lib.logictools.waveform.**draw_wavedrom**(*data*)
> Display the waveform using the Wavedrom package.
>
> Users can call this method directly to draw any wavedrom data.
>
> Example usage:

```
>>> a = {
    'signal': [
        {'name': 'clk', 'wave': 'p.....|...'},
        {'name': 'dat', 'wave': 'x.345x|=.x',
                        'data': ['head', 'body', 'tail', 'data']},
        {'name': 'req', 'wave': '0.1..0|1.0'},
        {},
        {'name': 'ack', 'wave': '1.....|01.'}
    ]}
>>> draw_wavedrom(a)
```

More information can be found at: https://github.com/witchard/nbwavedrom

> **Parameters data** (*dict*) – A dictionary of data as shown in the example.

pynq.lib.logictools.waveform.**int_to_sample**(*bits*)
   Function to convert a bit list into a multi-bit sample.

   Example: [1, 1, 1, 0] will be converted to 7, since the LSB of the sample appears first in the sequence.

   > **Parameters bits** (*list*) – A list of bits, each element being 0 or 1.

   > **Returns** A numpy uint32 converted from the bit samples.

   > **Return type** int

pynq.lib.logictools.waveform.**wave_to_bitstring**(*wave*)
   Function to convert a pattern consisting of *l*, *h*, and dot to a sequence of *0* and *1*.

   > **Parameters wave** (*str*) – The input string to convert.

   > **Returns** A bit sequence of 0's and 1's.

   > **Return type** str

## pynq.lib.pmod Package

The pynq.lib.pmod package is a collection of drivers for controlling peripherals attached to a PMOD port. The PMOD interface can control PMOD peripherals or Grove peripherals (via the PYNQ Grove shield)

## pynq.lib.pmod.pmod_adc Module

**class** pynq.lib.pmod.pmod_adc.**Pmod_ADC**(*mb_info*)
   Bases: object

   This class controls an Analog to Digital Converter Pmod.

   The Pmod AD2 (PB 200-217) is an analog-to-digital converter powered by AD7991. Users may configure up to 4 conversion channels at 12 bits of resolution.

   **microblaze**
      Microblaze processor instance used by this module.

      > **Type** Pmod

   **log_running**
      The state of the log (0: stopped, 1: started).

      > **Type** int

---

**get_log**()
> Get the log of voltage values.
>
> First stop the log before getting the log.
>
>> **Returns** List of voltage samples from the ADC.
>>
>> **Return type** list

**get_log_raw**()
> Get the log of raw values.
>
> First stop the log before getting the log.
>
>> **Returns** List of raw samples from the ADC.
>>
>> **Return type** list

**read**(*ch1=1*, *ch2=0*, *ch3=0*)
> Get the voltage from the Pmod ADC.
>
> When ch1, ch2, and ch3 values are 1 then the corresponding channel is included.
>
> For each channel selected, this method reads and returns one sample.

---

**Note:** The 4th channel is not available due to the jumper setting on ADC.

---

---

**Note:** This method reads the voltage values from ADC.

---

> **Parameters**
>
> - **ch1** (*int*) – 1 means include channel 1, 0 means do not include.
>
> - **ch2** (*int*) – 1 means include channel 2, 0 means do not include.
>
> - **ch3** (*int*) – 1 means include channel 3, 0 means do not include.
>
> **Returns** The voltage values read from the 3 channels of the Pmod ADC.
>
> **Return type** list

**read_raw**(*ch1=1*, *ch2=0*, *ch3=0*)
> Get the raw value from the Pmod ADC.
>
> When ch1, ch2, and ch3 values are 1 then the corresponding channel is included.
>
> For each channel selected, this method reads and returns one sample.

---

**Note:** The 4th channel is not available due to the jumper (JP1) setting on ADC.

---

---

**Note:** This method reads the raw value from ADC.

---

> **Parameters**
>
> - **ch1** (*int*) – 1 means include channel 1, 0 means do not include.
>
> - **ch2** (*int*) – 1 means include channel 2, 0 means do not include.

- **ch3** (*int*) – 1 means include channel 3, 0 means do not include.

> **Returns** The raw values read from the 3 channels of the Pmod ADC.

> **Return type** list

**reset**()
    Reset the ADC.

> **Returns**

> **Return type** None

**start_log**(*ch1=1*, *ch2=0*, *ch3=0*, *log_interval_us=100*)
    Start the log of voltage values with the interval specified.

    This parameter *log_interval_us* can set the time interval between two samples, so that users can read out multiple values in a single log.

> **Parameters**

>> - **ch1** (*int*) – 1 means include channel 1, 0 means do not include.

>> - **ch2** (*int*) – 1 means include channel 2, 0 means do not include.

>> - **ch3** (*int*) – 1 means include channel 3, 0 means do not include.

>> - **log_interval_us** (*int*) – The length of the log in milliseconds, for debug only.

> **Returns**

> **Return type** None

**start_log_raw**(*ch1=1*, *ch2=0*, *ch3=0*, *log_interval_us=100*)
    Start the log of raw values with the interval specified.

    This parameter *log_interval_us* can set the time interval between two samples, so that users can read out multiple values in a single log.

> **Parameters**

>> - **ch1** (*int*) – 1 means include channel 1, 0 means do not include.

>> - **ch2** (*int*) – 1 means include channel 2, 0 means do not include.

>> - **ch3** (*int*) – 1 means include channel 3, 0 means do not include.

>> - **log_interval_us** (*int*) – The length of the log in milliseconds, for debug only.

> **Returns**

> **Return type** None

**stop_log**()
    Stop the log of voltage values.

    This is done by sending the reset command to IOP. There is no need to wait for the IOP.

> **Returns**

> **Return type** None

**stop_log_raw**()
    Stop the log of raw values.

    This is done by sending the reset command to IOP. There is no need to wait for the IOP.

> **Returns**

> **Return type** None

## pynq.lib.pmod.pmod_als Module

**class** `pynq.lib.pmod.pmod_als.`**`Pmod_ALS`**(*mb_info*)

> Bases: `object`
>
> This class controls a light sensor Pmod.
>
> The Digilent Pmod ALS demonstrates light-to-digital sensing through a single ambient light sensor. This is based on an ADC081S021 analog-to-digital converter and a TEMT6000X01 ambient light sensor.
>
> **microblaze**
>
> > Microblaze processor instance used by this module.
> >
> > > **Type** Pmod
>
> **log_interval_ms**
>
> > Time in milliseconds between sampled reads.
> >
> > > **Type** int
>
> **get_log**()
>
> > Return list of logged samples.
> >
> > > **Returns**
> > >
> > > **Return type** List of valid samples from the ALS sensor [0-255]
>
> **read**()
>
> > Read current light value measured by the ALS Pmod.
> >
> > > **Returns** The current sensor value.
> > >
> > > **Return type** int
>
> **set_log_interval_ms**(*log_interval_ms*)
>
> > Set the length of the log in the ALS Pmod.
> >
> > This method can set the length of the log, so that users can read out multiple values in a single log.
> >
> > > **Parameters** **log_interval_ms** (*int*) – The length of the log in milliseconds, for debug only.
> > >
> > > **Returns**
> > >
> > > **Return type** None
>
> **start_log**()
>
> > Start recording multiple values in a log.
> >
> > This method will first call set_log_interval_ms() before sending the command.
> >
> > > **Returns**
> > >
> > > **Return type** None
>
> **stop_log**()
>
> > Stop recording multiple values in a log.
> >
> > Simply send the command to stop the log.
> >
> > > **Returns**
> > >
> > > **Return type** None

**pynq.lib.pmod.pmod_cable Module**

**class** pynq.lib.pmod.pmod_cable.**Pmod_Cable**(*mb_info*, *index*, *direction*, *cable*)

   Bases: *pynq.lib.pmod.pmod_io.Pmod_IO*

   This class can be used for a cable connecting Pmod interfaces.

   This class inherits from the Pmod IO class.

   ---

   **Note:**   When 2 Pmods are connected using a cable, the parameter 'cable' decides whether the cable is a 'loopback' or 'straight' cable. The default is a straight cable (no internal wire twisting). For pin mapping, please check the Pmod IO class.

   ---

   **microblaze**
      Microblaze processor instance used by this module.

         **Type**  Pmod

   **index**
      The index of the Pmod pin, from 0 to 7.

         **Type**  int

   **direction**
      Input 'in' or output 'out'.

         **Type**  str

   **cable**
      Either 'straight' or 'loopback'.

         **Type**  str

   **read**()
      Receive the value from the Pmod cable.

      This method overrides the read() method in the Pmod IO class. There are no new *write()* method defined in this class, so the *read()* will be inherited from Pmod IO class.

      ---

      **Note:**  Only use this function when direction = 'in'.

      When two Pmods are connected on the same board, for any received raw value, a "straignt" cable flips the upper 4 pins and the lower 4 pins: A Pmod interface <=> Another Pmod interface {vdd,gnd,3,2,1,0} <=> {vdd,gnd,7,6,5,4} {vdd,gnd,7,6,5,4} <=> {vdd,gnd,3,2,1,0}

      A "loop-back" cable satisfies the following mapping between two Pmods: A Pmod interface <=> Another Pmod interface {vdd,gnd,3,2,1,0} <=> {vdd,gnd,3,2,1,0} {vdd,gnd,7,6,5,4} <=> {vdd,gnd,7,6,5,4}

      ---

         **Returns**  The data (0 or 1) on the specified Pmod IO pin.

         **Return type**  int

   **set_cable**(*cable*)
      Set the type for the cable.

      ---

      **Note:**  The default cable type is 'straight'. Only straight cable or loop-back cable can be recognized.

      ---

> **Parameters cable** (*str*) – Either 'straight' or 'loopback'.
>
> **Returns**
>
> **Return type** None

### pynq.lib.pmod.pmod_dac Module

**class** `pynq.lib.pmod.pmod_dac.`**`Pmod_DAC`**(*mb_info*, *value=None*)

   Bases: `object`

   This class controls a Digital to Analog Converter Pmod.

   The Pmod DA4 (PB 200-245) is an 8 channel 12-bit digital-to-analog converter run via AD5628.

   **microblaze**

   > Microblaze processor instance used by this module.
   >
   > **Type** Pmod

   **write**(*value*)

   > Write a floating point number onto the DAC Pmod.

   ---

   **Note:** Input value must be in the range [0.00, 2.50]

   ---

   > **Parameters value** (*float*) – The value to be written to the DAC.
   >
   > **Returns**
   >
   > **Return type** None

### pynq.lib.pmod.pmod_dpot Module

**class** `pynq.lib.pmod.pmod_dpot.`**`Pmod_DPOT`**(*mb_info*)

   Bases: `object`

   This class controls a digital potentiometer Pmod.

   The Pmod DPOT (PB 200-239) is a digital potentiometer powered by the AD5160. Users may set a desired resistance between 60 ~ 10k ohms.

   **microblaze**

   > Microblaze processor instance used by this module.
   >
   > **Type** Pmod

   **write**(*val*, *step=0*, *log_ms=0*)

   > Write the value into the DPOT.
   >
   > This method will write the parameters "value", "step", and "log_ms" all together into the DPOT Pmod. The parameter "log_ms" is only used for debug; users can ignore this parameter.
   >
   > **Parameters**
   >
   > - **val** (*int*) – The initial value to start, in [0, 255].
   > - **step** (*int*) – The number of steps when ramping up to the final value.
   > - **log_ms** (*int*) – The length of the log in milliseconds, for debug only.

**Returns**

**Return type** None

## pynq.lib.pmod.pmod_iic Module

**class** pynq.lib.pmod.pmod_iic.**Pmod_IIC**(*mb_info*, *scl_pin*, *sda_pin*, *iic_addr*)
    Bases: pynq.lib.pmod.pmod_devmode.Pmod_DevMode

    This class controls the Pmod IIC pins.

---

**Note:** The index of the Pmod pins: upper row, from left to right: {vdd,gnd,3,2,1,0}. lower row, from left to right: {vdd,gnd,7,6,5,4}.

---

**microblaze**
        Microblaze processor instance used by this module.

        **Type** Pmod

**scl_pin**
        The SCL pin number.

        **Type** int

**sda_pin**
        The SDA pin number.

        **Type** int

**iic_addr**
        The IIC device address.

        **Type** int

**sr_addr**
        The IIC device SR address (base address + 0x104).

        **Type** int

**dtr_addr**
        The IIC device DTR address (base address + 0x108).

        **Type** int

**cr_addr**
        The IIC device CR address (base address + 0x100).

        **Type** int

**rfd_addr**
        The IIC device RFD address (base address + 0x120).

        **Type** int

**drr_addr**
        The IIC device DRR address (base address + 0x10C).

        **Type** int

**receive**(*num_bytes*)
        This method receives IIC bytes from the device.

> **Parameters num_bytes** (*int*) – Number of bytes to be received from the device.

> **Returns iic_bytes** – A list of 8-bit bytes received from the driver.

> **Return type** list

> **Raises** `RuntimeError` – Timeout when waiting for the RX FIFO to fill.

**send**(*iic_bytes*)

This method sends the command or data to the driver.

> **Parameters iic_bytes** (*list*) – A list of 8-bit bytes to be sent to the driver.

> **Returns**

> **Return type** None

> **Raises** `RuntimeError` – Timeout when waiting for the FIFO to be empty.

## pynq.lib.pmod.pmod_io Module

**class** pynq.lib.pmod.pmod_io.**Pmod_IO**(*mb_info*, *index*, *direction*)

Bases: `pynq.lib.pmod.pmod_devmode.Pmod_DevMode`

This class controls the Pmod IO pins as inputs or outputs.

---

**Note:** The parameter 'direction' determines whether the instance is input/output: 'in' : receiving input from offchip to onchip. 'out' : sending output from onchip to offchip. The index of the Pmod pins: upper row, from left to right: {vdd,gnd,3,2,1,0}. lower row, from left to right: {vdd,gnd,7,6,5,4}.

---

**microblaze**

Microblaze processor instance used by this module.

> **Type** Pmod

**index**

The index of the Pmod pin, starting from 0.

> **Type** int

**direction**

Input 'in' or output 'out'.

> **Type** str

**read**()

Receive the value from the offboard Pmod IO device.

---

**Note:** Only use this function when direction is 'in'.

---

> **Returns** The data (0 or 1) on the specified Pmod IO pin.

> **Return type** int

**write**(*value*)

Send the value to the offboard Pmod IO device.

---

**Note:** Only use this function when direction is 'out'.

---

> **Parameters value** (*int*) – The value to be written to the Pmod IO device.
>
> **Returns**
>
> **Return type** None

## pynq.lib.pmod.pmod_led8 Module

**class** `pynq.lib.pmod.pmod_led8.`**`Pmod_LED8`**(*mb_info*, *index*)

> Bases: `pynq.lib.pmod.pmod_devmode.Pmod_DevMode`
>
> This class controls a single LED on the LED8 Pmod.
>
> The Pmod LED8 (PB 200-163) has eight high-brightness LEDs. Each LED can be individually illuminated from a logic high signal.
>
> **microblaze**
> > Microblaze processor instance used by this module.
> >
> > **Type** Pmod
>
> **iop_switch_config**
> > Microblaze processor IO switch configuration (8 integers).
> >
> > **Type** list
>
> **index**
> > Index of the pin on LED8, starting from 0.
> >
> > **Type** int
>
> **off**()
> > Turn off a single LED.
> >
> > **Returns**
> >
> > **Return type** None
>
> **on**()
> > Turn on a single LED.
> >
> > **Returns**
> >
> > **Return type** None
>
> **read**()
> > Retrieve the LED state.
> >
> > **Returns** The data (0 or 1) read out from the selected pin.
> >
> > **Return type** int
>
> **toggle**()
> > Flip the bit of a single LED.

---

**Note:** The LED will be turned off if it is on. Similarly, it will be turned on if it is off.

---

**Returns**

**Return type** None

**write**(*value*)
Set the LED state according to the input value

---

**Note:** This method does not take into account the current LED state.

---

**Parameters value** (*int*) – Turn on the LED if value is 1; turn it off if value is 0.

**Returns**

**Return type** None

## pynq.lib.pmod.pmod_oled Module

**class** pynq.lib.pmod.pmod_oled.**Pmod_OLED**(*mb_info*, *text=None*)
Bases: `object`

This class controls an OLED Pmod.

The Pmod OLED (PB 200-222) is 128x32 pixel monochrome organic LED (OLED) panel powered by the Solomon Systech SSD1306.

**microblaze**
Microblaze processor instance used by this module.

**Type** Pmod

**clear**()
Clear the OLED screen.

This is done by sending the clear command to the IOP.

**Returns**

**Return type** None

**draw_line**(*x1*, *y1*, *x2*, *y2*)
Draw a straight line on the OLED.

**Parameters**

- **x1** (*int*) – The x-position of the starting point.

- **y1** (*int*) – The y-position of the starting point.

- **x2** (*int*) – The x-position of the ending point.

- **y2** (*int*) – The y-position of the ending point.

**Returns**

**Return type** None

**draw_rect**(*x1*, *y1*, *x2*, *y2*)
Draw a rectangle on the OLED.

**Parameters**

- **x1** (*int*) – The x-position of the starting point.

---

- **y1** (*int*) – The y-position of the starting point.
- **x2** (*int*) – The x-position of the ending point.
- **y2** (*int*) – The y-position of the ending point.

**Returns**

**Return type** None

**write** (*text*, *x=0*, *y=0*)
    Write a new text string on the OLED.

**Parameters**

- **text** (*str*) – The text string to be displayed on the OLED screen.
- **x** (*int*) – The x-position of the display.
- **y** (*int*) – The y-position of the display.

**Returns**

**Return type** None

## pynq.lib.pmod.pmod_pwm Module

**class** pynq.lib.pmod.pmod_pwm.**Pmod_PWM**(*mb_info*, *index*)
    Bases: object

This class uses the PWM of the IOP.

**microblaze**
    Microblaze processor instance used by this module.

    **Type** Pmod

**generate** (*period*, *duty_cycle*)
    Generate pwm signal with desired period and percent duty cycle.

**Parameters**

- **period** (*int*) – The period of the tone (us), between 1 and 65536.
- **duty_cycle** (*int*) – The duty cycle in percentage.

**Returns**

**Return type** None

**stop** ()
    Stops PWM generation.

**Returns**

**Return type** None

## pynq.lib.pmod.pmod_tc1 Module

**class** pynq.lib.pmod.pmod_tc1.**Pmod_TC1**(*mb_info*)
    Bases: object

This class controls a thermocouple Pmod.

The Digilent PmodTC1 is a cold-junction thermocouple-to-digital converter module designed for a classic K-Type thermocouple wire. With Maxim Integrated's MAX31855, this module reports the measured temperature in 14-bits with 0.25 degC resolution.

**microblaze**
>   Microblaze processor instance used by this module.

>>   **Type** Pmod

**log_interval_ms**
>   Time in milliseconds between sampled reads.

>>   **Type** int

**get_log**()
>   Return list of logged samples.

---

>   **Note:** The logged samples are raw 32-bit samples captured from the sensor.

---

>>   **Returns** List of valid samples from the TC1 sensor

>>   **Return type** list

**read_alarm_flags**()
>   Read the alarm flags from the raw value.

>>   **Returns** The alarm flags from the TC1. bit 0 = 1 if thermocouple connection is open-circuit; bit 1 = 1 if thermocouple connection is shorted to generated; bit 2 = 1 if thermocouple connection is shorted to VCC; bit 16 = 1 if any if bits 0-2 are 1.

>>   **Return type** u32

**read_junction_temperature**()
>   Read the reference junction temperature.

>>   **Returns** The reference junction temperature in degC.

>>   **Return type** float

**read_raw**()
>   Read full 32-bit register of TC1 Pmod.

>>   **Returns** The current register contents.

>>   **Return type** int

**read_thermocouple_temperature**()
>   Read the reference junction temperature.

>>   **Returns** The thermocouple temperature in degC.

>>   **Return type** float

**set_log_interval_ms**(*log_interval_ms*)
>   Set the length of the log in the TC1 Pmod.

>   This method can set the length of the log, so that users can read out multiple values in a single log.

>>   **Parameters** **log_interval_ms** (*int*) – The length of the log in milliseconds, for debug only.

>>   **Returns**

> **Return type** None

**start_log**()
> Start recording multiple values in a log.

> This method will first call set_log_interval_ms() before writting to the MMIO.

> > **Returns**

> > **Return type** None

**stop_log**()
> Stop recording multiple values in a log.

> Simply write to the MMIO to stop the log.

> > **Returns**

> > **Return type** None

pynq.lib.pmod.pmod_tc1.**reg_to_alarms**(*reg_val*)
> Extracts Alarm flags from 32-bit register value.

> > **Parameters** **reg_val** (*int*) – 32-bit TC1 register value

> > **Returns** The alarm flags from the TC1. bit 0 = 1 if thermocouple connection is open-circuit; bit 1 = 1 if thermocouple connection is shorted to generated; bit 2 = 1 if thermocouple connection is shorted to VCC; bit 16 = 1 if any if bits 0-2 are 1.

> > **Return type** u32

pynq.lib.pmod.pmod_tc1.**reg_to_ref**(*reg_val*)
> Extracts Ref Junction temperature from 32-bit register value.

> > **Parameters** **reg_val** (*int*) – 32-bit TC1 register value

> > **Returns** The reference junction temperature in degC.

> > **Return type** float

pynq.lib.pmod.pmod_tc1.**reg_to_tc**(*reg_val*)
> Extracts Thermocouple temperature from 32-bit register value.

> > **Parameters** **reg_val** (*int*) – 32-bit TC1 register value

> > **Returns** The thermocouple temperature in degC.

> > **Return type** float

## pynq.lib.pmod.pmod_timer Module

**class** pynq.lib.pmod.pmod_timer.**Pmod_Timer**(*mb_info*, *index*)
> Bases: object

> This class uses the timer's capture and generation capabilities.

> **microblaze**
> > Microblaze processor instance used by this module.

> > > **Type** Pmod

> **clk_period_ns**
> > The clock period of the IOP in ns.

> > > **Type** int

**event_count** (*period*)

> Count the number of rising edges detected in (period) clocks.

> > **Parameters** **period** (*int*) – The period of the generated signals.

> > **Returns** The number of events detected.

> > **Return type** int

**event_detected** (*period*)

> Detect a rising edge or high-level in (period) clocks.

> > **Parameters** **period** (*int*) – The period of the generated signals.

> > **Returns** 1 if any event is detected, and 0 if no event is detected.

> > **Return type** int

**generate_pulse** (*period*, *times=0*)

> Generate pulses every (period) clocks for a number of times.

> The default is to generate pulses every (period) IOP clocks forever until stopped. The pulse width is equal to the IOP clock period.

> > **Parameters**

> > > • **period** (*int*) – The period of the generated signals.

> > > • **times** (*int*) – The number of times for which the pulses are generated.

> > **Returns**

> > **Return type** None

**get_period_ns** ()

> Measure the period between two successive rising edges.

> > **Returns** Measured period in ns.

> > **Return type** int

**stop** ()

> This method stops the timer.

> > **Returns**

> > **Return type** None

## pynq.lib.pmod.pmod_tmp2 Module

**class** pynq.lib.pmod.pmod_tmp2.**Pmod_TMP2** (*mb_info*)

> Bases: object

> This class controls a temperature sensor Pmod.

> The Pmod TMP2 (PB 200-221) is an ambient temperature sensor powered by ADT7420.

> **microblaze**

> > Microblaze processor instance used by this module.

> > **Type** Pmod

> **log_interval_ms**

> > Time in milliseconds between sampled reads.

> > **Type** int

**get_log**()
> Return list of logged samples.

>> **Returns**

>> **Return type** List of valid samples from the temperature sensor in Celsius.

**read**()
> Read current temperature value measured by the Pmod TMP2.

>> **Returns** The current sensor value.

>> **Return type** float

**set_log_interval_ms**(*log_interval_ms*)
> Set the sampling interval for the Pmod TMP2.

>> **Parameters** **log_interval_ms** (*int*) – Time in milliseconds between sampled reads of the TMP2 sensor

>> **Returns**

>> **Return type** None

**start_log**()
> Start recording multiple values in a log.

> This method will first call set_log_interval_ms() before writting to the MMIO.

>> **Returns**

>> **Return type** None

**stop_log**()
> Stop recording multiple values in a log.

> Simply write to the MMIO to stop the log.

>> **Returns**

>> **Return type** None

## pynq.lib.pmod.pmod_grove_adc Module

**class** pynq.lib.pmod.pmod_grove_adc.**Grove_ADC**(*mb_info*, *gr_pin*)
> Bases: object

> This class controls the Grove IIC ADC.

> Grove ADC is a 12-bit precision ADC module based on ADC121C021. Hardware version: v1.2.

**microblaze**
> Microblaze processor instance used by this module.

>> **Type** Pmod

**log_running**
> The state of the log (0: stopped, 1: started).

>> **Type** int

**log_interval_ms**
> Time in milliseconds between sampled reads.

>> **Type** int

**get_log**()
>    Return list of logged samples.

>>    **Returns**  List of valid voltage samples (floats) from the ADC sensor.

>>    **Return type**  list

**get_log_raw**()
>    Return list of logged raw samples.

>>    **Returns**  List of valid raw samples from the ADC sensor.

>>    **Return type**  list

**read**()
>    Read the ADC voltage from the Grove ADC peripheral.

>>    **Returns**  The float value after translation.

>>    **Return type**  float

**read_raw**()
>    Read the ADC raw value from the Grove ADC peripheral.

>>    **Returns**  The raw value from the sensor.

>>    **Return type**  int

**reset**()
>    Resets/initializes the ADC.

>>    **Returns**

>>    **Return type**  None

**set_log_interval_ms**(*log_interval_ms*)
>    Set the length of the log for the Grove ADC peripheral.

>    This method can set the time interval between two samples, so that users can read out multiple values in a single log.

>>    **Parameters**  **log_interval_ms** (*int*) – The time between two samples in milliseconds, for logging only.

>>    **Returns**

>>    **Return type**  None

**start_log**()
>    Start recording multiple voltage values (float) in a log.

>    This method will first call set_log_interval_ms() before sending the command.

>>    **Returns**

>>    **Return type**  None

**start_log_raw**()
>    Start recording raw data in a log.

>    This method will first call set_log_interval_ms() before sending the command.

>>    **Returns**

>>    **Return type**  None

**stop_log**()
> Stop recording the voltage values in the log.

> Simply send the command 0xC to stop the log.

> > **Returns**

> > **Return type** None

**stop_log_raw**()
> Stop recording the raw values in the log.

> Simply send the command 0xC to stop the log.

> > **Returns**

> > **Return type** None

## pynq.lib.pmod.pmod_grove_buzzer Module

**class** pynq.lib.pmod.pmod_grove_buzzer.**Grove_Buzzer**(*mb_info*, *gr_pin*)
> Bases: object

> This class controls the Grove Buzzer.

> The grove buzzer module has a piezo buzzer as the main component. The piezo can be connected to digital outputs, and will emit a tone when the output is HIGH. Alternatively, it can be connected to an analog pulse-width modulation output to generate various tones and effects. Hardware version: v1.2.

> **microblaze**
> > Microblaze processor instance used by this module.

> > **Type** Pmod

> **play_melody**()
> > Play a melody.

> > > **Returns**

> > > **Return type** None

> **play_tone**(*tone_period*, *num_cycles*)
> > Play a single tone with tone_period for num_cycles

> > > **Parameters**

> > > - **tone_period** (*int*) – The period of the tone in microsecond.

> > > - **num_cycles** (*int*) – The number of cycles for the tone to be played.

> > > **Returns**

> > > **Return type** None

## pynq.lib.pmod.pmod_grove_dlight Module

**class** pynq.lib.pmod.pmod_grove_dlight.**Grove_Dlight**(*mb_info*, *gr_pin*)
> Bases: object

> This class controls the Grove IIC color sensor.

> Grove Color sensor based on the TCS3414CS. Hardware version: v1.3.

**microblaze**
Microblaze processor instance used by this module.

> **Type** Pmod

**read_lux**()
Read the computed lux value of the sensor.

> **Returns** The lux value from the sensor

> **Return type** int

**read_raw_light**()
Read the visible and IR channel values.

Read the values from the grove digital light peripheral.

> **Returns** A tuple containing 2 integer values ch0 (visible) and ch1 (IR).

> **Return type** tuple

## pynq.lib.pmod.pmod_grove_ear_hr Module

**class** `pynq.lib.pmod.pmod_grove_ear_hr`.**Grove_EarHR**(*mb_info*, *gr_pin*)
Bases: `object`

This class controls the Grove ear clip heart rate sensor.

Sensor model: MED03212P.

**microblaze**
Microblaze processor instance used by this module.

> **Type** Pmod

**read**()
Read the heart rate from the sensor.

> **Returns** The heart rate as beats per minute

> **Return type** float

**read_raw**()
Read the number of heart beats.

Read the number of beats since the sensor initialization; also read the time elapsed in ms between the latest two heart beats.

> **Returns** Number of heart beats and the time elapsed between 2 latest beats.

> **Return type** tuple

## pynq.lib.pmod.pmod_grove_finger_hr Module

**class** `pynq.lib.pmod.pmod_grove_finger_hr`.**Grove_FingerHR**(*mb_info*, *gr_pin*)
Bases: `object`

This class controls the Grove finger clip heart rate sensor.

Grove Finger sensor based on the TCS3414CS. Hardware version: v1.3.

**microblaze**
Microblaze processor instance used by this module.

> **Type** Pmod

**log_running**
>  The state of the log (0: stopped, 1: started).

>> **Type** int

**log_interval_ms**
>  Time in milliseconds between sampled reads.

>> **Type** int

**get_log()**
>  Return list of logged samples.

>> **Returns** List of integers containing the heart rate.

>> **Return type** list

**read()**
>  Read the heart rate value from the Grove Finger HR peripheral.

>> **Returns** An integer representing the heart rate frequency.

>> **Return type** int

**start_log**(*log_interval_ms=100*)
>  Start recording multiple heart rate values in a log.

>  This method will first call set the log interval before writing to the MMIO.

>> **Parameters** **log_interval_ms** (*int*) – The time between two samples in milliseconds.

>> **Returns**

>> **Return type** None

**stop_log()**
>  Stop recording the values in the log.

>  Simply write 0xC to the MMIO to stop the log.

>> **Returns**

>> **Return type** None

## pynq.lib.pmod.pmod_grove_haptic_motor Module

**class** pynq.lib.pmod.pmod_grove_haptic_motor.**Grove_HapticMotor**(*mb_info*, *gr_pin*)
>  Bases: object

>  This class controls the Grove Haptic Motor based on the DRV2605L. Hardware version v0.9.

**microblaze**
>  Microblaze processor instance used by this module.

>> **Type** Pmod

**is_playing()**
>  Check if a vibration effect is running on the motor.

>> **Returns** True if a vibration effect is playing, false otherwise

>> **Return type** bool

**play**(*effect*)

> Play a vibration effect on the Grove Haptic Motor peripheral.
>
> Valid effect identifiers are in the range [1, 127].
>
> > **Parameters effect** (*int*) – An integer that specifies the effect.
> >
> > **Returns**
> >
> > **Return type** None

**play_sequence**(*sequence*)

> Play a sequence of effects possibly separated by pauses.
>
> At most 8 effects or pauses can be specified at a time. Pauses are defined using negative integer values in the range [-1, -127] that correspond to a pause length in the range [10, 1270] ms
>
> Valid effect identifiers are in the range [1, 127]
>
> As an example, in the following sequence example: [4,-20,5] effect 4 is played and after a pause of 200 ms effect 5 is played
>
> > **Parameters sequence** (*list*) – At most 8 values specifying effects and pauses.
> >
> > **Returns**
> >
> > **Return type** None

**stop**()

> Stop an effect or a sequence on the motor peripheral.
>
> > **Returns**
> >
> > **Return type** None

### pynq.lib.pmod.pmod_grove_imu Module

**class** pynq.lib.pmod.pmod_grove_imu.**Grove_IMU**(*mb_info*, *gr_pin*)

> Bases: object
>
> This class controls the Grove IIC IMU.
>
> Grove IMU 10DOF is a combination of grove IMU 9DOF (MPU9250) and grove barometer sensor (BMP180). MPU-9250 is a 9-axis motion tracking device that combines a 3-axis gyroscope, 3-axis accelerometer, 3-axis magnetometer and a Digital Motion Processor (DMP). BMP180 is a high precision, low power digital pressure sensor. Hardware version: v1.1.

**microblaze**

> Microblaze processor instance used by this module.
>
> > **Type** Pmod

**get_accl**()

> Get the data from the accelerometer.
>
> > **Returns** A list of the acceleration data along X-axis, Y-axis, and Z-axis.
> >
> > **Return type** list

**get_altitude**()

> Get the current altitude.
>
> > **Returns** The altitude value.
> >
> > **Return type** float

**get_atm** ()
> Get the current pressure in relative atmosphere.
>
> > **Returns** The related atmosphere.
> >
> > **Return type** float

**get_compass** ()
> Get the data from the magnetometer.
>
> > **Returns** A list of the compass data along X-axis, Y-axis, and Z-axis.
> >
> > **Return type** list

**get_gyro** ()
> Get the data from the gyroscope.
>
> > **Returns** A list of the gyro data along X-axis, Y-axis, and Z-axis.
> >
> > **Return type** list

**get_heading** ()
> Get the value of the heading.
>
> > **Returns** The angle deviated from the X-axis, toward the positive Y-axis.
> >
> > **Return type** float

**get_pressure** ()
> Get the current pressure in Pa.
>
> > **Returns** The pressure value.
> >
> > **Return type** float

**get_temperature** ()
> Get the current temperature in degree C.
>
> > **Returns** The temperature value.
> >
> > **Return type** float

**get_tilt_heading** ()
> Get the value of the tilt heading.
>
> > **Returns** The tilt heading value.
> >
> > **Return type** float

**reset** ()
> Reset all the sensors on the grove IMU.
>
> > **Returns**
> >
> > **Return type** None

## pynq.lib.pmod.pmod_grove_ledbar Module

**class** pynq.lib.pmod.pmod_grove_ledbar.**Grove_LEDbar** (*mb_info*, *gr_pin*)
> Bases: object

This class controls the Grove LED BAR.

Grove LED Bar is comprised of a 10 segment LED gauge bar and an MY9221 LED controlling chip. Model: LED05031P. Hardware version: v2.0.

---

**microblaze**
> Microblaze processor instance used by this module.
>
> > **Type** Pmod

**read**()
> Reads the current status of LEDbar.
>
> Reads the current status of LED bar and returns 10-bit binary string. Each bit position corresponds to a LED position in the LEDbar, and bit value corresponds to the LED state.
>
> Red LED corresponds to the LSB, while green LED corresponds to the MSB.
>
> > **Returns** String of 10 binary bits.
> >
> > **Return type** str

**reset**()
> Resets the LEDbar.
>
> Clears the LED bar, sets all LEDs to OFF state.
>
> > **Returns**
> >
> > **Return type** None

**write_binary**(*data_in*)
> Set individual LEDs in the LEDbar based on 10 bit binary input.
>
> Each bit in the 10-bit *data_in* points to a LED position on the LEDbar. Red LED corresponds to the LSB, while green LED corresponds to the MSB.
>
> > **Parameters** **data_in** (*int*) – 10 LSBs of this parameter control the LEDbar.
> >
> > **Returns**
> >
> > **Return type** None

**write_brightness**(*data_in*, *brightness=[170, 170, 170, 170, 170, 170, 170, 170, 170, 170]*)
> Set individual LEDs with 3 level brightness control.
>
> Each bit in the 10-bit *data_in* points to a LED position on the LEDbar. Red LED corresponds to the LSB, while green LED corresponds to the MSB.
>
> Brightness of each LED is controlled by the brightness parameter. There are 3 perceivable levels of brightness: 0xFF : HIGH 0xAA : MED 0x01 : LOW
>
> > **Parameters**
> >
> > > • **data_in** (*int*) – 10 LSBs of this parameter control the LEDbar.
> > >
> > > • **brightness** (*list*) – Each element controls a single LED.
> >
> > **Returns**
> >
> > **Return type** None

**write_level**(*level*, *bright_level*, *green_to_red*)
> Set the level to which the leds are to be lit in levels 1 - 10.
>
> Level can be set in both directions. *set_level* operates by setting all LEDs to the same brightness level.
>
> There are 4 preset brightness levels: bright_level = 0: off bright_level = 1: low bright_level = 2: medium bright_level = 3: maximum
>
> *green_to_red* indicates the direction, either from red to green when it is 0, or green to red when it is 1.
>
> > **Parameters**

- **level** (*int*) – 10 levels exist, where 1 is minimum and 10 is maximum.

- **bright_level** (*int*) – Controls brightness of all LEDs in the LEDbar, from 0 to 3.

- **green_to_red** (*int*) – Sets the direction of the sequence.

**Returns**

**Return type** None

### pynq.lib.pmod.pmod_grove_light Module

**class** pynq.lib.pmod.pmod_grove_light.**Grove_Light**(*mb_info*, *gr_pin*)

Bases: *pynq.lib.pmod.pmod_grove_adc.Grove_ADC*

This class controls the grove light sensor.

This class inherits from the grove ADC class. To use this module, grove ADC has to be used as a bridge. The light sensor incorporates a Light Dependent Resistor (LDR) GL5528. Hardware version: v1.1.

**microblaze**

Microblaze processor instance used by this module.

> **Type** Pmod

**log_running**

The state of the log (0: stopped, 1: started).

> **Type** int

**log_interval_ms**

Time in milliseconds between sampled reads.

> **Type** int

**get_log**()

Return list of logged light sensor resistances.

> **Returns** List of valid light sensor resistances.

> **Return type** list

**read**()

Read the light sensor resistance in from the light sensor.

This method overrides the definition in grove ADC.

> **Returns** The light reading in terms of the sensor resistance.

> **Return type** float

**start_log**()

Start recording the light sensor resistance in a log.

This method will call the start_log_raw() in the parent class.

> **Returns**

> **Return type** None

**stop_log**()

Stop recording light values in a log.

This method will call the stop_log_raw() in the parent class.

> **Returns**

> **Return type** None

### pynq.lib.pmod.pmod_grove_oled Module

**class** pynq.lib.pmod.pmod_grove_oled.**Grove_OLED**(*mb_info*, *gr_pin*)

> Bases: `object`
>
> This class controls the Grove IIC OLED.
>
> Grove LED 128×64 Display module is an OLED monochrome 128×64 matrix display module. Model: OLE35046P. Hardware version: v1.1.
>
> **microblaze**
>
> > Microblaze processor instance used by this module.
> >
> > > **Type** Pmod
>
> **clear**()
>
> > Clear the OLED screen.
> >
> > This is done by writing empty strings into the OLED in Microblaze.
> >
> > > **Returns**
> > >
> > > **Return type** None
>
> **set_contrast**(*brightness*)
>
> > Set the contrast level for the OLED display.
> >
> > The contrast level is in [0, 255].
> >
> > > **Parameters** **brightness** (*int*) – The brightness of the display.
> > >
> > > **Returns**
> > >
> > > **Return type** None
>
> **set_horizontal_mode**()
>
> > Set the display mode to horizontal.
> >
> > > **Returns**
> > >
> > > **Return type** None
>
> **set_inverse_mode**()
>
> > Set the display mode to inverse.
> >
> > > **Returns**
> > >
> > > **Return type** None
>
> **set_normal_mode**()
>
> > Set the display mode to normal.
> >
> > > **Returns**
> > >
> > > **Return type** None
>
> **set_page_mode**()
>
> > Set the display mode to paged.
> >
> > > **Returns**
> > >
> > > **Return type** None

**set_position**(*row*, *column*)
:   Set the position of the display.

    The position is indicated by (row, column).

    > **Parameters**
    >
    > - **row** (*int*) – The row number to start the display.
    >
    > - **column** (*int*) – The column number to start the display.
    >
    > **Returns**
    >
    > **Return type** None

**write**(*text*)
:   Write a new text string on the OLED.

    Clear the screen first to correctly show the new text.

    > **Parameters** **text** (*str*) – The text string to be displayed on the OLED screen.
    >
    > **Returns**
    >
    > **Return type** None

## pynq.lib.pmod.pmod_grove_pir Module

**class** pynq.lib.pmod.pmod_grove_pir.**Grove_PIR**(*mb_info*, *gr_pin*)
:   Bases: *pynq.lib.pmod.pmod_io.Pmod_IO*

    This class controls the PIR motion sensor.

    Hardware version: v1.2.

    **microblaze**
    :   Microblaze processor instance used by this module.

        > **Type** Pmod

    **index**
    :   The index of the Pmod pin, from 0 to 7.

        > **Type** int

    **direction**
    :   Can only be 'in' for PIR sensor.

        > **Type** str

    **read**()
    :   Receive the value from the PIR sensor.

        Returns 0 when there is no motion, and returns 1 otherwise.

        > **Returns** The data (0 or 1) read from the PIR sensor.
        >
        > **Return type** int

## pynq.lib.pmod.pmod_grove_th02 Module

**class** pynq.lib.pmod.pmod_grove_th02.**Grove_TH02**(*mb_info*, *gr_pin*)
:   Bases: object

This class controls the Grove I2C Temperature and Humidity sensor.

Temperature & humidity sensor (high-accuracy & mini). Hardware version: v1.0.

**microblaze**
> Microblaze processor instance used by this module.
>
> > **Type** Pmod

**log_running**
> The state of the log (0: stopped, 1: started).
>
> > **Type** int

**log_interval_ms**
> Time in milliseconds between sampled reads.
>
> > **Type** int

**get_log**()
> Return list of logged samples.
>
> > **Returns** List of tuples containing (temperature, humidity)
> >
> > **Return type** list

**read**()
> Read the temperature and humidity values from the TH02 peripheral.
>
> > **Returns** Tuple containing (temperature, humidity)
> >
> > **Return type** tuple

**start_log**(*log_interval_ms=100*)
> Start recording multiple heart rate values in a log.
>
> This method will first call set the log interval before sending the command.
>
> > **Parameters** **log_interval_ms** (*int*) – The time between two samples in milliseconds.
> >
> > **Returns**
> >
> > **Return type** None

**stop_log**()
> Stop recording the values in the log.
>
> Simply send the command 0xC to stop the log.
>
> > **Returns**
> >
> > **Return type** None

### pynq.lib.pmod.pmod_grove_tmp Module

**class** pynq.lib.pmod.pmod_grove_tmp.**Grove_TMP**(*mb_info*, *gr_pin*, *version='v1.2'*)
> Bases: *pynq.lib.pmod.pmod_grove_adc.Grove_ADC*

This class controls the grove temperature sensor.

This class inherits from the Grove_ADC class. To use this module, grove ADC has to be used as a bridge. The temperature sensor uses a thermistor to detect the ambient temperature. Hardware version: v1.2.

**microblaze**
> Microblaze processor instance used by this module.

> **Type** Pmod

**log_running**
>    The state of the log (0: stopped, 1: started).

>    > **Type** int

**log_interval_ms**
>    Time in milliseconds between sampled reads.

>    > **Type** int

**bValue**
>    The thermistor constant.

>    > **Type** int

**get_log**()
>    Return list of logged temperature samples.

>    > **Returns** List of valid temperature readings from the temperature sensor.

>    > **Return type** list

**read**()
>    Read temperature values in Celsius from temperature sensor.

>    This method overrides the definition in Grove_ADC.

>    > **Returns** The temperature reading in Celsius.

>    > **Return type** float

**start_log**()
>    Start recording temperature in a log.

>    This method will call the start_log_raw() in the parent class.

**stop_log**()
>    Stop recording temperature in a log.

>    This method will call the stop_log_raw() in the parent class.

>    > **Returns**

>    > **Return type** None

## pynq.lib.pynqmicroblaze.pynqmicroblaze Module

**class** pynq.lib.pynqmicroblaze.pynqmicroblaze.**MBInterruptEvent**(*intr_pin*,
                                                                        *intr_ack_gpio*)
>    Bases: object

>    The class provides and asyncio Event-like interface to the interrupt subsystem for a Microblaze. The event is set by raising an interrupt and cleared using the clear function.

>    Typical use is to call clear prior to sending a request to the Microblaze and waiting in a loop until the response is received. This order of operations will avoid race conditions between the Microblaze and the host code.

>    **clear**()
>    >    Clear the interrupt and reset the event. Resetting the event should be done before sending a request that will be acknowledged interrupts.

>    **wait**()
>    >    Coroutine to wait until the event is set by an interrupt.

**class** pynq.lib.pynqmicroblaze.pynqmicroblaze.**MicroblazeHierarchy**(*description*,
                                                                              *mb-*
                                                                              *type='Unknown'*)

    Bases: *pynq.overlay.DefaultHierarchy*

    Hierarchy driver for the microblaze subsystem.

    Enables the user to *load* programs on to the microblaze. All function calls and member accesses are delegated
    to the loaded program.

    **static checkhierarchy**(*description*)
        Function to check if the driver matches a particular hierarchy

        This function should be redefined in derived classes to return True if the description matches what is
        expected by the driver. The default implementation always returns False so that drivers that forget don't
        get loaded for hierarchies they don't expect.

    **mbtype**
        The defined type of the microblaze subsystem. Used by driver programs to limit what microblaze subsys-
        tems the program is run on.

**class** pynq.lib.pynqmicroblaze.pynqmicroblaze.**PynqMicroblaze**(*mb_info*,
                                                                       *mb_program*,
                                                                       *force=False*)

    Bases: object

    This class controls the active Microblaze instances in the system.

    **ip_name**
        The name of the IP corresponding to the Microblaze.

        **Type** str

    **rst_name**
        The name of the reset pin for the Microblaze.

        **Type** str

    **mb_program**
        The absolute path of the Microblaze program.

        **Type** str

    **state**
        The status (IDLE, RUNNING, or STOPPED) of the Microblaze.

        **Type** str

    **reset_pin**
        The reset pin associated with the Microblaze.

        **Type** *GPIO*

    **mmio**
        The MMIO instance associated with the Microblaze.

        **Type** *MMIO*

    **interrupt**
        An asyncio.Event-like class for waiting on and clearing interrupts.

        **Type** Event

    **program**()
        This method programs the Microblaze.

This method is called in __init__(); it can also be called after that. It uses the attribute *self.mb_program* to program the Microblaze.

> **Returns**

> **Return type** None

**read**(*offset*, *length=1*)

This method reads data from the shared memory of Microblaze.

> **Parameters**

> - **offset** (`int`) – The beginning offset where data are read from.

> - **length** (`int`) – The number of data (32-bit int) to be read.

> **Returns** An int of a list of data read from the shared memory.

> **Return type** int/list

**reset**()

Reset the Microblaze to stop it from running.

This method will update the status of the Microblaze.

> **Returns**

> **Return type** None

**run**()

Start the Microblaze to run program loaded.

This method will update the status of the Microblaze.

> **Returns**

> **Return type** None

**write**(*offset*, *data*)

This method write data into the shared memory of the Microblaze.

> **Parameters**

> - **offset** (`int`) – The beginning offset where data are written into.

> - **data** (`int/list`) – A list of 32b words to be written.

> **Returns**

> **Return type** None

## pynq.lib.pynqmicroblaze.compile Module

**class** pynq.lib.pynqmicroblaze.compile.**MicroblazeProgram**(*mb_info*, *program_text*, *bsp=None*)

> Bases: *pynq.lib.pynqmicroblaze.pynqmicroblaze.PynqMicroblaze*

pynq.lib.pynqmicroblaze.compile.**dependencies**(*source*, *bsp*)

pynq.lib.pynqmicroblaze.compile.**preprocess**(*source*, *bsp=None*, *mb_info=None*)

**pynq.lib.pynqmicroblaze.rpc Module**

**class** `pynq.lib.pynqmicroblaze.rpc.`**`ConstPointerWrapper`**(*type_*, *struct_string*)
    Bases: `object`

    Wrapper for const T pointers, transfers data in only one direction.

    **`param_decode`**(*old_val*, *stream*)

    **`param_encode`**(*old_val*)

    **`post_argument`**(*name*)

    **`pre_argument`**(*name*)

    **`return_decode`**(*stream*)

**class** `pynq.lib.pynqmicroblaze.rpc.`**`FuncAdapter`**(*decl*, *typedefs*)
    Bases: `object`

    Provides the C and Python interfaces for a function declaration

    **`return_interface`**
        The type wrapper for the return type

            **Type**  TypeWrapper

    **`arg_interfaces`**
        An array of type wrappers for the arguments

            **Type**  [TypeWrapper]

    **`call_ast`**
        Syntax tree for the wrapped function call

            **Type**  pycparser.c_ast

    **`pack_args`**(*\*args*)
        Create a bytes of the provided arguments

    **`receive_response`**(*stream*, *\*args*)
        Reads the response stream, updates arguments and returns the value of the function call if applicable

**class** `pynq.lib.pynqmicroblaze.rpc.`**`FuncDefVisitor`**
    Bases: `pycparser.c_ast.NodeVisitor`

    Primary visitor that parses out function definitions, typedes and enumerations from a syntax tree

    **`visit_Enum`**(*node*)

    **`visit_FuncDecl`**(*node*)

    **`visit_FuncDef`**(*node*)

    **`visit_Typedef`**(*node*)

**class** `pynq.lib.pynqmicroblaze.rpc.`**`MicroblazeFunction`**(*stream*, *index*, *function*, *return_type*)
    Bases: `object`

    Calls a specific function

    **`call_async`**(*\*args*)

**class** `pynq.lib.pynqmicroblaze.rpc.`**`MicroblazeLibrary`**(*iop*, *libraries*)
    Bases: *`pynq.lib.pynqmicroblaze.rpc.MicroblazeRPC`*

    Provides simple Python-only access to a set of Microblaze libraries.

The members of this class are determined by the libraries chosen and can determined either by using `dir` on the instance or the `?` operator inside of IPython

**class** pynq.lib.pynqmicroblaze.rpc.**MicroblazeRPC**(*iop*, *program_text*)
    Bases: `object`

    Provides a python interface to the Microblaze based on an RPC mechanism.

    The attributes of the class are generated dynamically from the typedefs, enumerations and functions given in the provided source.

    Functions are added as methods, the values in enumerations are added as constants to the class and types are added as classes.

    **release**()
        Alias for *reset()*

    **reset**()
        Reset and free the microblaze for use by other programs

**class** pynq.lib.pynqmicroblaze.rpc.**ParsedEnum**
    Bases: `object`

    Holds the values of an enum from the C source

**class** pynq.lib.pynqmicroblaze.rpc.**PointerWrapper**(*type_*, *struct_string*)
    Bases: `object`

    Wrapper for non-const T pointers that retrieves any data modified by the called function.

    **param_decode**(*old_val*, *stream*)

    **param_encode**(*old_val*)

    **post_argument**(*name*)

    **pre_argument**(*name*)

    **return_decode**(*stream*)

**class** pynq.lib.pynqmicroblaze.rpc.**PrimitiveWrapper**(*struct_string*, *type_*)
    Bases: `object`

    Wrapper for C primitives that can be represented by a single Struct string.

    **param_decode**(*old_val*, *stream*)

    **param_encode**(*old_val*)

    **post_argument**(*name*)

    **pre_argument**(*name*)

    **return_decode**(*stream*)

**class** pynq.lib.pynqmicroblaze.rpc.**VoidPointerWrapper**(*type_*)
    Bases: `object`

    Wrapper for a void* pointer that will refer to a physically contiguous chunk of memory.

    **param_decode**(*old_val*, *stream*)

    **param_encode**(*old_val*)

    **post_argument**(*name*)

    **pre_argument**(*name*)

> **return_decode**(*stream*)

**class** pynq.lib.pynqmicroblaze.rpc.**VoidWrapper**

> Bases: object
>
> Wraps void - only valid for return types
>
> **param_decode**(*old_val*, *stream*)
>
> **param_encode**(*old_val*)
>
> **post_argument**(*name*)
>
> **pre_argument**(*name*)
>
> **return_decode**(*stream*)

### pynq.lib.pynqmicroblaze.magic Module

**class** pynq.lib.pynqmicroblaze.magic.**MicroblazeMagics**(*shell=None*, *\*\*kwargs*)

> Bases: IPython.core.magic.Magics
>
> **magics = {'cell': {'microblaze': 'microblaze'}, 'line': {}}**
>
> **microblaze**(*line*, *cell*)
>
> **name2obj**(*name*)
>
> **registered = True**

### pynq.lib.pynqmicroblaze.streams Module

**class** pynq.lib.pynqmicroblaze.streams.**InterruptMBStream**(*iop*, *read_offset=62464*, *write_offset=61440*)

> Bases: *pynq.lib.pynqmicroblaze.streams.SimpleMBStream*
>
> **read_async**()
>
> **wait_for_data_async**()

**class** pynq.lib.pynqmicroblaze.streams.**SimpleMBChannel**(*buffer*, *offset=0*, *length=0*)

> Bases: object
>
> **buffer_space**()
>
> **bytes_available**()
>
> **read**(*n=-1*)
>
> **read_upto**(*n=-1*)
>
> **write**(*b*)

**class** pynq.lib.pynqmicroblaze.streams.**SimpleMBStream**(*iop*, *read_offset=62464*, *write_offset=61440*)

> Bases: object
>
> **buffer_space**()
>
> **bytes_available**()
>
> **read**(*n=-1*)
>
> **read_byte**()

**read_float** ()

**read_int16** ()

**read_int32** ()

**read_string** ()

**read_uint16** ()

**read_uint32** ()

**write** (*b*)

**write_address** (*p*, *adjust=True*)

**write_byte** (*b*)

**write_float** (*f*)

**write_int16** (*i*)

**write_int32** (*i*)

**write_string** (*s*)

**write_uint16** (*u*)

**write_uint32** (*u*)

## pynq.lib.pynqmicroblaze.bsp Module

**class** `pynq.lib.pynqmicroblaze.bsp.`**BSPInstance** (*root*)
    Bases: `object`

**class** `pynq.lib.pynqmicroblaze.bsp.`**Module** (*root*)
    Bases: `object`

`pynq.lib.pynqmicroblaze.bsp.`**add_bsp** (*directory*)

`pynq.lib.pynqmicroblaze.bsp.`**add_module_path** (*directory*)

## pynq.lib.rgbled Module

The pynq.lib.rgbled module is a driver for controlling onboard Red-Green-Blue (RGB) Light Emitting Diodes (LEDs).

**class** `pynq.lib.rgbled.`**RGBLED** (*index*)
    Bases: `object`

    This class controls the onboard RGB LEDs.

    **index**
        The index of the RGB LED, from 4 (LD4) to 5 (LD5).

            **Type** int

    **_mmio**
        Shared memory map for the RGBLED GPIO controller.

            **Type** *MMIO*

    **_rgbleds_val**
        Global value of the RGBLED GPIO pins.

            **Type** int

---

**off**()
> Turn off a single RGBLED.

>> **Returns**

>> **Return type**  None

**on**(*color*)
> Turn on a single RGB LED with a color value (see color constants).

>> **Parameters  color** (*int*) – Color of RGB specified by a 3-bit RGB integer value.

>> **Returns**

>> **Return type**  None

**read**()
> Retrieve the RGBLED state.

>> **Returns**  The color value stored in the RGBLED.

>> **Return type**  int

**write**(*color*)
> Set the RGBLED state according to the input value.

>> **Parameters  color** (*int*) – Color of RGB specified by a 3-bit RGB integer value.

>> **Returns**

>> **Return type**  None

## pynq.lib.rpi Package

The pynq.lib.rpi package is a collection of drivers for controlling peripherals attached to a RPi interface.  The RPi interface can control Raspberry Pi peripherals.

## pynq.lib.rpi Module

**class** pynq.lib.rpi.rpi.**Rpi**(*mb_info*, *mb_program*)
> Bases: *pynq.lib.pynqmicroblaze.pynqmicroblaze.PynqMicroblaze*

> This class controls the Raspberry Pi Microblaze instances in the system.

> This class inherits from the PynqMicroblaze class. It extends PynqMicroblaze with capability to control Raspberry Pi devices.

> **ip_name**
>> The name of the IP corresponding to the Microblaze.

>> **Type**  str

> **rst_name**
>> The name of the reset pin for the Microblaze.

>> **Type**  str

> **mb_program**
>> The absolute path of the Microblaze program.

>> **Type**  str

**state**
>    The status (IDLE, RUNNING, or STOPPED) of the Microblaze.

>    >    **Type** str

**reset_pin**
>    The reset pin associated with the Microblaze.

>    >    **Type** *GPIO*

**mmio**
>    The MMIO instance associated with the Microblaze.

>    >    **Type** *MMIO*

**interrupt**
>    An asyncio.Event-like class for waiting on and clearing interrupts.

>    >    **Type** Event

**read_mailbox**(*data_offset*, *num_words=1*)
>    This method reads mailbox data from the Microblaze.

>    >    **Parameters**

>    >    >    • **data_offset** (*int*) – The offset for mailbox data, 0,4,… for MAILBOX 0,1,…

>    >    >    • **num_words** (*int*) – Number of 32b words to read from Microblaze mailbox.

>    >    **Returns** An int of a list of data read from the mailbox.

>    >    **Return type** int/list

**write_blocking_command**(*command*)
>    This method writes a blocking command to the Microblaze.

>    The program waits in the loop until the command is cleared by the Microblaze.

>    >    **Parameters command** (*int*) – The command to write to the Microblaze.

>    >    **Returns**

>    >    **Return type** None

**write_mailbox**(*data_offset*, *data*)
>    This method write data into the mailbox of the Microblaze.

>    >    **Parameters**

>    >    >    • **data_offset** (*int*) – The offset for mailbox data, 0,4,… for MAILBOX 0,1,…

>    >    >    • **data** (*int/list*) – A list of 32b words to be written into the mailbox.

>    >    **Returns**

>    >    **Return type** None

**write_non_blocking_command**(*command*)
>    This method writes a non-blocking command to the Microblaze.

>    The program will just send the command and returns the control immediately.

>    >    **Parameters command** (*int*) – The command to write to the Microblaze.

>    >    **Returns**

>    >    **Return type** None

**pynq.lib.switch Module**

The pynq.lib.switch module is a driver for reading, and waiting for value changes on onboard switches.

**class** pynq.lib.switch.**Switch**(*device*)

> Bases: `object`
>
> This class controls the onboard switches.
>
> **_impl**
> > An object with appropriate Switch methods
> >
> > > **Type** object
>
> **read**()
> > Read the current value of the switch.
>
> **wait_for_value**(*value*)
> > Wait for the switch to be closed or opened.
> >
> > > **Parameters** **value** (`int`) – 1 for the switch up and 0 for the switch down

**pynq.lib.video Module**

The pynq.lib.video module is a driver capturing streaming HDMI input, producing streaming HDMI output and hardware-accelerated colorspace conversion.

**pynq.lib.video.clocks Module**

**class** pynq.lib.video.clocks.**DP159**(*master*, *address*)

> Bases: `object`
>
> Class to configure the TI SNDP159 HDMI redriver/retimer
>
> **set_clock**(*refclk*, *line_rate*)
> > Configure the device based on the line rate

**class** pynq.lib.video.clocks.**IDT_8T49N24**(*master*, *address*)

> Bases: `object`
>
> Driver for the IDT 8T49N24x series of clock generators
>
> **check_device_id**()
>
> **enable**(*active*)
>
> **set_clock**(*freq*, *line_rate*)

**pynq.lib.video.common Module**

**class** pynq.lib.video.common.**PixelFormat**(*bits_per_pixel*, *in_color*, *out_color*, *fourcc=None*)

> Bases: `object`
>
> Wrapper for all of the information about a video format
>
> **bits_per_pixel**
> > Number of bits for each pixel
> >
> > > **Type** int

**in_color**
> Coefficients from BGR stream to pixel format

>> **Type** list of float

**out_color**
> Coefficient from pixel format to BGR stream

>> **Type** list of float

**class** pynq.lib.video.common.**VideoMode**(*width*, *height*, *bits_per_pixel*, *fps=60*, *stride=None*)
> Bases: object

> Class for holding the information about a video mode

**height**
> Height of the video frame in lines

>> **Type** int

**width**
> Width of the video frame in pixels

>> **Type** int

**stride**
> Width of a line in the video frame in bytes

>> **Type** int

**bits_per_pixel**
> Bits per pixel

>> **Type** int

**bytes_per_Pixel**
> Bytes required to represent each pixel

>> **Type** int

**shape**
> Numpy-style tuple describing the video frame

>> **Type** tuple of int

## pynq.lib.video.dma Module

**class** pynq.lib.video.dma.**AxiVDMA**(*description*, *framecount=4*)
> Bases: *pynq.overlay.DefaultIP*

> Driver class for the Xilinx VideoDMA IP core

> The driver is split into input and output channels are exposed using the readchannel and writechannel attributes. Each channel has start and stop methods to control the data transfer. All channels MUST be stopped before reprogramming the bitstream or inconsistent behaviour may result.

> The DMA uses a single ownership model of frames in that frames are either owned by the DMA or the user code but not both. S2MMChannel.readframe and MM2SChannel.newframe both return a frame to the user. It is the user's responsibility to either free the frame using the freebuffer() method or to hand ownership back to the DMA using MM2SChannel.writeframe. Once ownership has been returned the user should not access the contents of the frame as the underlying memory may be deleted without warning.

**readchannel**
>   Video input DMA channel

>>      **Type** *AxiVDMA.S2MMChannel*

**writechannel**
>   Video output DMA channel

>>      **Type** *AxiVDMA.MM2SChannel*

**class MM2SChannel**(*parent*, *interrupt*)
>   Bases: `object`

>   DMA channel from memory to a video output.

>   Will continually repeat the most recent frame written.

>   **mode**
>>      Video mode of the DMA channel
>>>         **Type** *VideoMode*

>   **cacheable_frames**
>>      Whether frames should be stored in cacheable or non-cacheable memory
>>>         **Type** bool

>   **activeframe**

>   **desiredframe**

>   **framedelay**

>   **mode**
>>      The video mode of the DMA, must be called prior to starting. If changed while the DMA channel is running the channel will be stopped

>   **newframe**()
>>      Returns a frame of the appropriate size for the video mode.

>>      The contents of the frame are undefined and should not be assumed to be black
>>>         **Returns**
>>>         **Return type** numpy.ndarray video frame

>   **parked**
>>      Is the channel parked or running in circular buffer mode

>   **reload**()
>>      Reload the configuration of the DMA. Should only be called by the _FrameList class or if you really know what you are doing

>   **reset**()
>>      Soft reset the DMA channel

>   **running**

>   **setframe**(*frame*)
>>      Sets a frame without blocking or taking ownership. In most circumstances writeframe() is more appropriate

>   **start**()
>>      Start the DMA channel with a blank screen. The mode must be set prior to calling or a RuntimeError will result.

>   **stop**()
>>      Stop the DMA channel and empty the frame cache

**writeframe**(*frame*)

> Schedule the specified frame to be the next one displayed. Assumes ownership of frame which should no longer be modified by the user. May block if there is already a frame scheduled.

**writeframe_async**(*frame*)

> Same as writeframe() but yields instead of blocking if a frame is already scheduled

**class S2MMChannel**(*parent*, *interrupt*)

> Bases: `object`
>
> Read channel of the Video DMA
>
> Brings frames from the video input into memory. Hands ownership of the read frames to the user code.
>
> **mode**
>
> > The video mode of the DMA channel
> > > **Type** *VideoMode*
>
> **cacheable_frames**
>
> > Whether frames should be stored in cacheable or non-cacheable memory
> > > **Type** bool
>
> **activeframe**
>
> > The frame index currently being processed by the DMA
> >
> > This process requires clearing any error bits in the DMA channel
>
> **desiredframe**
>
> > The next frame index to the processed by the DMA
>
> **irqframecount**
>
> **mode**
>
> > The video mode of the DMA. Must be set prior to starting. Changing this while the DMA is running will result in the DMA being stopped.
>
> **parked**
>
> > Is the channel parked or running in circular buffer mode
>
> **readframe**()
>
> > Read a frame from the channel and return to the user
> >
> > This function may block until a complete frame has been read. A single frame buffer is kept so the first frame read after a long pause in reading may return a stale frame. To ensure an up-to-date frame when starting processing video read an additional time before starting the processing loop.
> > > **Returns**
> > > **Return type** numpy.ndarray of the video frame
>
> **readframe_async**()
>
> > Read a frame from the channel, yielding instead of blocking if no data is available. See readframe for more details
>
> **reload**()
>
> > Reload the configuration of the DMA. Should only be called by the _FrameList class or if you really know what you are doing
>
> **reset**()
>
> > Soft reset the DMA. Finishes all transfers before starting the reset process
>
> **running**
>
> > Is the DMA channel running

**start**()
> Start the DMA. The mode must be set prior to this being called

**stop**()
> Stops the DMA, clears the frame cache and unhooks any tied outputs

**tie**(*channel*)
> Ties an output channel to this input channel. This is used to pass video from input to output without invoking the CPU for each frame. Main use case is when some slow processing is being done on a subset of frames while the video is passed through directly to the output. Only one output may be tied to an output. The tie is broken either by calling tie(None) or writing a frame to the tied output channel.

**bindto = ['xilinx.com:ip:axi_vdma:6.2', 'xilinx.com:ip:axi_vdma:6.3']**

## pynq.lib.video.drm Module

**class** pynq.lib.video.drm.**DrmDriver**(*device_path*, *event_loop=None*)
> Bases: `object`

> Driver for DRM-based output through the Linux kernel

> This driver provides a zero-copy interface to the DRM subsystem exposing a similar API to the HDMI interface. The API should be configured with a PixelFormat containing a FourCC which will be passed to the Linux video subsystem.

> Once configured frames can be retrieved using *newframe* which returns a numpy array mapped to a frame buffer. The frame can be switched using *writeframe*. Once a frame has been written it should not be modified as ownership has been transferred to the kernel.

**close**()
> Close the display device

**configure**(*mode*, *pixelformat*)
> Configure the display output

> Raises an exception if the initialisation fails.

> **Parameters**
>> - **mode** (`VideoMode`) – The resolution to set the output display to
>> - **pixelformat** (`PixelFormat`) – The pixel format to use - must contain a fourcc

**newframe**()
> Return a new frame which can later be written

> Frames are not zeroed before being returned so the calling application should make sure the frame is fully written.

> **Returns** pynq.PynqBuffer

> **Return type** numpy.ndarray mapped to a hardware frame

**return_pointer**(*pointer*)

**start**()
> Dummy function to match the HDMI interface

**stop**()
> Dummy function to match the HDMI interface

> **writeframe**(*frame*)
>> Write a frame to the display.
>>
>> Raises an exception if the operation fails and blocks until a page-flip if there is already a frame scheduled to be displayed.
>>
>>> **Parameters frame** (*pynq.ContiguousArray*) – Frame to write - must have been created by *newframe*
>
> **writeframe_async**(*frame*)
>> Write a frame to the display.
>>
>> Raises an exception if the operation fails and yields until a page-flip if there is already a frame scheduled to be displayed.
>>
>>> **Parameters frame** (*pynq.ContiguousArray*) – Frame to write - must have been created by *newframe*

## pynq.lib.video.dvi Module

**class** `pynq.lib.video.dvi.`**HDMIInFrontend**(*description*)

> Bases: *pynq.lib.video.frontend.VideoInFrontend*, *pynq.overlay.DefaultHierarchy*
>
> Class for interacting the with HDMI input frontend
>
> This class is used for enabling the HDMI input and retrieving the mode of the incoming video
>
> **mode**
>> The detected mode of the incoming video stream
>>
>>> **Type** *VideoMode*
>
> **static checkhierarchy**(*description*)
>> Function to check if the driver matches a particular hierarchy
>>
>> This function should be redefined in derived classes to return True if the description matches what is expected by the driver. The default implementation always returns False so that drivers that forget don't get loaded for hierarchies they don't expect.
>
> **mode**
>
> **start**(*init_timeout=60*)
>> Method that blocks until the video mode is successfully detected
>
> **stop**()
>> Currently empty function included for symmetry with the HDMIOutFrontend class

**class** `pynq.lib.video.dvi.`**HDMIOutFrontend**(*description*)

> Bases: *pynq.lib.video.frontend.VideoOutFrontend*, *pynq.overlay.DefaultHierarchy*
>
> Class for interacting the HDMI output frontend
>
> This class is used for enabling the HDMI output and setting the desired mode of the video stream
>
> **mode**
>> Desired mode for the output video. Must be set prior to calling start
>>
>>> **Type** *VideoMode*
>
> **static checkhierarchy**(*description*)
>> Function to check if the driver matches a particular hierarchy

This function should be redefined in derived classes to return True if the description matches what is expected by the driver. The default implementation always returns False so that drivers that forget don't get loaded for hierarchies they don't expect.

**mode**
> Get or set the video mode for the HDMI output, must be set to one of the following resolutions:

> 640x480 800x600 1280x720 1280x1024 1920x1080

> Any other resolution will result in a ValueError being raised. The bits per pixel will always be 24 when retrieved and ignored when set.

**start = None**
> Start the HDMI output - requires the that mode is already set

**stop = None**
> Stop the HDMI output

## pynq.lib.video.frontend Module

**class** pynq.lib.video.frontend.**VideoInFrontend**(*args*, ***kwargs*)
> Bases: object

**class** pynq.lib.video.frontend.**VideoOutFrontend**(*args*, ***kwargs*)
> Bases: object

## pynq.lib.video.hierarchies Module

**class** pynq.lib.video.hierarchies.**HDMIWrapper**(*description*)
> Bases: *pynq.overlay.DefaultHierarchy*

Hierarchy driver for the entire video subsystem.

Exposes the input, output and video DMA as attributes. For most use cases the wrappers for the input and output pipelines are sufficient and the VDMA will not need to be used directly.

**hdmi_in**
> The HDMI input pipeline

> > **Type** pynq.lib.video.HDMIIn

**hdmi_out**
> The HDMI output pipeline

> > **Type** pynq.lib.video.HDMIOut

**axi_vdma**
> The video DMA.

> > **Type** pynq.lib.video.AxiVDMA

**static checkhierarchy**(*description*)
> Function to check if the driver matches a particular hierarchy

> This function should be redefined in derived classes to return True if the description matches what is expected by the driver. The default implementation always returns False so that drivers that forget don't get loaded for hierarchies they don't expect.

**class** pynq.lib.video.hierarchies.**VideoIn**(*description*, *vdma=None*)
    Bases: *pynq.overlay.DefaultHierarchy*

    Wrapper for the input video pipeline.

    This wrapper assumes the following pipeline structure and naming

    color_convert_in -> pixel_pack ->axi_vdma with vtc_in and axi_gpio_hdmiiin helper IP

    **frontend**
        The HDMI frontend for signal detection

            **Type** pynq.lib.video.HDMIInFrontend

    **color_convert**
        The input color format converter

            **Type** pynq.lib.video.ColorConverter

    **pixel_pack**
        Converts the input pixel size to that required by the VDMA

            **Type** pynq.lib.video.PixelPacker

    **cacheable_frames**
        Whether frames should be cacheable or non-cacheable

        Only valid if a VDMA has been specified

    **static checkhierarchy**(*description*)
        Function to check if the driver matches a particular hierarchy

        This function should be redefined in derived classes to return True if the description matches what is expected by the driver. The default implementation always returns False so that drivers that forget don't get loaded for hierarchies they don't expect.

    **close**()
        Uninitialise the drivers, stopping the pipeline beforehand

    **colorspace**
        The colorspace of the pipeline, can be changed without stopping the pipeline

    **configure**(*pixelformat=<pynq.lib.video.common.PixelFormat object>*)
        Configure the pipeline to use the specified pixel format.

        If the pipeline is running it is stopped prior to the configuration being changed

            **Parameters pixelformat** (*PixelFormat*) – The pixel format to configure the pipeline for

    **mode**
        Video mode of the input

    **readframe**()
        Read a video frame

        See AxiVDMA.S2MMChannel.readframe for details

    **readframe_async**()
        Read a video frame

        See AxiVDMA.S2MMChannel.readframe for details

    **start**()
        Start the pipeline

**stop**()
> Stop the pipeline

**tie**(*output*)
> Mirror the video input on to an output channel

> > **Parameters output** (*HDMIOut*) – The output to mirror on to

**class** pynq.lib.video.hierarchies.**VideoOut**(*description*, *vdma=None*)
> Bases: `pynq.overlay.DefaultHierarchy`

> Wrapper for the output video pipeline.

> This wrapper assumes the following pipeline structure and naming

> axi_vdma -> pixel_unpack -> color_convert -> frontend with vtc_out and axi_dynclk helper IP

> **frontend**
> > The HDMI frontend for mode setting

> > > **Type** pynq.lib.video.HDMIOutFrontend

> **color_convert**
> > The output color format converter

> > > **Type** pynq.lib.video.ColorConverter

> **pixel_unpack**
> > Converts the input pixel size to 24 bits-per-pixel

> > > **Type** pynq.lib.video.PixelPacker

> **cacheable_frames**
> > Whether frames should be cacheable or non-cacheable

> > Only valid if a VDMA has been specified

> **static checkhierarchy**(*description*)
> > Function to check if the driver matches a particular hierarchy

> > This function should be redefined in derived classes to return True if the description matches what is expected by the driver. The default implementation always returns False so that drivers that forget don't get loaded for hierarchies they don't expect.

> **close**()
> > Close the pipeline an unintialise the drivers

> **colorspace**
> > Set the colorspace for the pipeline - can be done without stopping the pipeline

> **configure**(*mode*, *pixelformat=None*)
> > Configure the pipeline to use the specified pixel format and size.

> > If the pipeline is running it is stopped prior to the configuration being changed

> > > **Parameters**

> > > - **mode** (*VideoMode*) – The video mode to output

> > > - **pixelformat** (*PixelFormat*) – The pixel format to configure the pipeline for

> **mode**
> > The currently configured video mode

> **newframe**()
> > Return an unintialised video frame of the correct type for the pipeline

**start**()
   Start the pipeline

**stop**()
   Stop the pipeline

**writeframe**(*frame*)
   Write the frame to the video output

   See AxiVDMA.MM2SChannel.writeframe for more details

**writeframe_async**(*frame*)
   Write the frame to the video output

   See AxiVDMA.MM2SChannel.writeframe for more details

## pynq.lib.video.pipeline Module

**class** pynq.lib.video.pipeline.**ColorConverter**(*description*)
   Bases: *pynq.overlay.DefaultIP*

   Driver for the color space converter

   The colorspace convert implements a 3x4 matrix for performing arbitrary linear color conversions. Each coefficient is represented as a 10 bit signed fixed point number with 2 integer bits. The result of the computation can visualised as a table

   # in1 in2 in3 1 # out1 c1 c2 c3 c10 # out2 c4 c5 c6 c11 # out3 c7 c8 c9 c12

   The color can be changed mid-stream.

   **colorspace**
      The coefficients of the colorspace conversion

         **Type** list of float

   **bindto = ['xilinx.com:hls:color_convert:1.0', 'xilinx.com:hls:color_convert_2:1.0']**

   **colorspace**
      The colorspace to convert. See the class description for details of the coefficients. The coefficients are a list of floats of length 12

**class** pynq.lib.video.pipeline.**PixelPacker**(*description*)
   Bases: *pynq.overlay.DefaultIP*

   Driver for the pixel format convert

   Changes the number of bits per pixel in the video stream. The stream should be paused prior to the width being changed. This can be targeted at either a pixel_pack or a pixel_unpack IP core.For a packer the input is always 24 bits per pixel while for an unpacker the output 24 bits per pixel.

   **bindto = ['xilinx.com:hls:pixel_pack:1.0', 'xilinx.com:hls:pixel_unpack:1.0', 'xilinx.**

   **bits_per_pixel**
      Number of bits per pixel in the stream

      Valid values are 8, 24 and 32. The following table describes the operation for packing and unpacking for each width

      Mode Pack Unpack 8 bpp Keep only the first channel Pad other channels with 0 16 bpp Dependent on resample Dependent on resample 24 bpp No change No change 32 bpp Pad channel 4 with 0 Discard channel 4

**resample**
> Perform chroma resampling in 16 bpp mode

> Boolean property that only affects 16 bpp mode. If True then the two chroma channels are multiplexed on to the second output pixel, otherwise only the first and second channels are transferred and the third is discarded

### pynq.lib.video.xilinx_hdmi Module

**class** pynq.lib.video.xilinx_hdmi.**HdmiRxSs**(*description*)
> Bases: *pynq.overlay.DefaultIP*, *pynq.lib.video.frontend.VideoInFrontend*

> Driver for the HDMI receiver subsystem

> **bindto = ['xilinx.com:ip:v_hdmi_rx_ss:3.1']**

> **load_edid**(*data*)
>> Configure the EDID data exposed by the receiver

>> The EDID should be between 128 and 256 bytes depending on the resolutions desired. In order to trigger the EDID to be read by the source the HPD line should be toggled after the EDID has been loaded.

>>> Parameters **data** (*bytes-like*) – EDID data to load

> **mode**
>> Return the mode of the attached device

> **report**()
>> Write the status of the receiver to stdout

> **set_hpd**(*value*)
>> Set the Host presence detect line

>> 1 or True advertises the presence of a monitor to the source 0 or False shows a disconnected cable

>>> Parameters **value** (*int or Boolean*) – The desired value of the HPD line

> **set_phy**(*phy*)
>> Set the attached PHY

>> The subsystem must be attached to a Xilinx HDMI PHY to operate

>>> Parameters **phy** (*HdmiVPhy*) – The driver for the PHY

> **start**()
>> Start the receiver

>> Blocks until the signal is stabilised

> **stop**()
>> Stop the receiver

**class** pynq.lib.video.xilinx_hdmi.**HdmiTxSs**(*description*)
> Bases: *pynq.overlay.DefaultIP*, *pynq.lib.video.frontend.VideoOutFrontend*

> Driver for the HDMI transmit subsystem

> **DviMode**()
>> Output using DVI framing

> **HdmiMode**()
>> Output using HDMI framing

> **bindto = ['xilinx.com:ip:v_hdmi_tx_ss:3.1']**

**handle_events**()
>   Ensure that interrupt handlers are called

**read_edid**()
>   Return the EDID of the attached monitor
>
>>   **Returns**  bytes
>>
>>   **Return type**  256 bytes of EDID data

**report**()
>   Write the status of the transmitter to stdout

**set_phy**(*phy*)
>   Set the attached PHY
>
>   The subsystem must be attached to a Xilinx HDMI PHY to operate
>
>>   **Parameters**  **phy** (*HdmiVPhy*) – The driver for the PHY

**start**()
>   Start the HDMI output
>
>   The mode attribute and the PHY of the driver must be set before the transmitter can be started.

**stop**()
>   Stop the HDMI transmitter

**wait_for_connect**()
>   Wait for a cable to connected to the transmitter port

**class** pynq.lib.video.xilinx_hdmi.**Vphy**(*description*)
>   Bases: *pynq.overlay.DefaultIP*
>
>   Driver for Xilinx HDMI PHY
>
>   **bindto = ['xilinx.com:ip:vid_phy_controller:2.2']**
>
>   **initialize**()
>
>   **report**()
>>   Write the status of the PHY to stdout

## pynq.lib.wifi Module

The pynq.lib.wifi module is a python module for interacting with WiFI adapters. This module can be used to connect and disconnect to wireless networks.

**class** pynq.lib.wifi.**Wifi**(*interface='wlan0'*)
>   Bases: object
>
>   This class controls the WiFi connection.
>
>   For USB WiFi, RALink RT5370 devices are recommended.

---

**Note:** Administrator rights are necessary to create network interface file

---

**wifi_port**
>   string identifier of the wireless network device
>
>>   **Type**  str

**connect** (*ssid*, *password*, *auto=False*)
> Make a new wireless connection.

> This function kills the wireless connection and connect to a new one using network ssid and WPA passphrase. Wrong ssid or passphrase will reject the connection.

> > **Parameters**
> > - **ssid** (`str`) – Unique identifier of the wireless network
> > - **password** (`str`) – String WPA passphrase necessary to access the network
> > - **auto** (`bool`) – Whether to set the interface as auto connected after boot.

> > **Returns**

> > **Return type** None

**gen_network_file** (*ssid*, *password*, *auto=False*)
> Generate the network authentication file.

> Generate the file from network SSID and WPA passphrase

> > **Parameters**
> > - **ssid** (`str`) – String unique identifier of the wireless network
> > - **password** (`str`) – String WPA passphrase necessary to access the network
> > - **auto** (`bool`) – Whether to set the interface as auto connected after boot.

> > **Returns**

> > **Return type** None

**reset** ()
> Shutdown the network connection.

> This function shutdown the network connection and delete the interface file.

> > **Returns**

> > **Return type** None

## 2.8.2 pynq.pl_server Package

The pynq.pl_server module contains the required modules for PL server. Each Linux device works as a singleton client to communicate with PL servers via a unique socket. Multiple devices can co-exist at the same time on a specific platform.

**Modules:**

- pynq.pl_server.server - Implements the top-level PL server for all the devices
- pynq.pl_server.device - Implements the (programmable) device class.
- pynq.pl_server.xrt_device - Implements Xilinx Run Time (XRT) enabled device.
- pynq.pl_server.hwh_parser - Parses useful information from hwh files.
- pynq.pl_server.tcl_parser - Parses useful information from tcl files.
- pynq.pl_server.xclbin_parser - Parses useful information from xclbin files.

### pynq.pl_server.server Module

The pynq.pl_server.server module manages all the device servers. The top-level PL server class manages multiple device servers, while each device server serves a unique communication socket for a programmable device. On embedded system, usually there is only one device per board; hence only one device server is managed by the PL server. The device client class eases the access to the device servers for users.

**class** `pynq.pl_server.server.`**`DeviceClient`**(*tag*, *key=b'xilinx'*)

> Bases: `object`
>
> Class to access the PL server
>
> The properties of the class access the most recent version from the PL server and are read-only. All updating of the PL server is performed by methods.
>
> **`bitfile_name`**
>> The getter for the attribute *bitfile_name*.
>>
>>> **Returns** The absolute path of the bitstream currently on PL.
>>>
>>> **Return type** str
>
> **`clear_devicetree`**()
>> Clear the device tree dictionary.
>>
>> This should be used when downloading the full bitstream, where all the dtbo are cleared from the system.
>
> **`clear_dict`**()
>> Clear all the dictionaries stored in PL.
>>
>> This method will clear all the related dictionaries, including IP dictionary, GPIO dictionary, etc.
>
> **`client_request`**()
>> Client connects to the PL server and receives the attributes.
>>
>> This method should not be used by the users directly. To check open pipes in the system, use *lsof | grep <address>* and *kill -9 <pid>* to manually delete them.
>>
>>> **Parameters**
>>>
>>> - **address** (*str*) – The filename on the file system.
>>> - **key** (*bytes*) – The authentication key of connection.
>>>
>>> **Returns**
>>>
>>> **Return type** None
>
> **`devicetree_dict`**
>
> **`gpio_dict`**
>> The getter for the attribute *gpio_dict*.
>>
>>> **Returns** The dictionary storing the PS GPIO pins.
>>>
>>> **Return type** dict
>
> **`hierarchy_dict`**
>> The getter for the attribute *hierarchy_dict*
>>
>>> **Returns** The dictionary containing the hierarchies in the design
>>>
>>> **Return type** dict

---

**insert_device_tree**(*abs_dtbo*)
    Insert device tree segment.

    For device tree segments associated with full / partial bitstreams, users can provide the relative or absolute paths of the dtbo files.

    > **Parameters** **abs_dtbo** (`str`) – The absolute path to the device tree segment.

**interrupt_controllers**
    The getter for the attribute *interrupt_controllers*.

    > **Returns** The dictionary storing interrupt controller information.

    > **Return type** dict

**interrupt_pins**
    The getter for the attribute *interrupt_pins*.

    > **Returns** The dictionary storing the interrupt endpoint information.

    > **Return type** dict

**ip_dict**
    The getter for the attribute *ip_dict*.

    > **Returns** The dictionary storing addressable IP instances; can be empty.

    > **Return type** dict

**load_ip_data**(*ip_name*, *data*)
    This method writes data to the addressable IP.

    ---

    **Note:** The data is assumed to be in binary format (.bin). The data name will be stored as a state information in the IP dictionary.

    ---

    > **Parameters**
    >
    > - **ip_name** (`str`) – The name of the addressable IP.
    > - **data** (`str`) – The absolute path of the data to be loaded.
    > - **zero** (`bool`) – Zero out the address of the IP not covered by data
    >
    > **Returns**
    >
    > **Return type** None

**mem_dict**
    The getter for the attribute *mem_dict*

    > **Returns** The dictionary containing the memory spaces in the design

    > **Return type** dict

**remove_device_tree**(*abs_dtbo*)
    Remove device tree segment for the overlay.

    > **Parameters** **abs_dtbo** (`str`) – The absolute path to the device tree segment.

**reset**(*parser=None*, *timestamp=None*, *bitfile_name=None*)
    Reset all the dictionaries.

This method must be called after a bitstream download. 1. In case there is a *hwh* or *tcl* file, this method will reset the states of the IP, GPIO, and interrupt dictionaries . 2. In case there is no *hwh* or *tcl* file, this method will simply clear the state information stored for all dictionaries.

An existing parser given as the input can significantly reduce the reset time, since the PL can reset based on the information provided by the parser.

> **Parameters** **parser** (*TCL/HWH*) – A parser object to speed up the reset process.

**server_update**(*continued=1*)

**timestamp**
    The getter for the attribute *timestamp*.

> **Returns** Bitstream download timestamp.

> **Return type** str

**update_partial_region**(*hier*, *parser*)
    Merge the parser information from partial region.

    Combine the currently PL information and the partial HWH/TCL file parsing results.

> **Parameters**
>
> - **hier** (*str*) – The name of the hierarchical block as the partial region.
>
> - **parser** (*TCL/HWH*) – A parser object for the partial region.

**class** pynq.pl_server.server.**DeviceServer**(*tag*, *key=b'xilinx'*)
    Bases: object

**server_proc**()

**start**()

**stop**()

**class** pynq.pl_server.server.**PLServer**
    Bases: object

**start**()

**stop**()

pynq.pl_server.server.**clear_state**(*dict_in*)
    Clear the state information for a given dictionary.

> **Parameters** **dict_in** (*dict*) – Input dictionary to be cleared.

## pynq.pl_server.device Module

The pynq.pl_server.device module exposes the user frontend for a device on a Linux system. This class, as a singleton for each device, manages the device status by communicating with the device server. As a special case, the XlnkDevice extends the device class and implements its own way of memory management.

**class** pynq.pl_server.device.**Device**(*tag*)
    Bases: object

**allocate**(*shape*, *dtype*, *\*\*kwargs*)
    Allocate an array on the device

    Returns a buffer on memory accessible to the device

> **Parameters**

- **shape** (*tuple(int)*) – The shape of the array

- **dtype** (*np.dtype*) – The type of the elements of the array

> **Returns** The buffer shared between the host and the device
>
> **Return type** *PynqBuffer*

**bitfile_name**

The getter for the attribute *bitfile_name*.

> **Returns** The absolute path of the bitstream currently on PL.
>
> **Return type** str

**clear_devicetree**()

Clear the device tree dictionary.

This should be used when downloading the full bitstream, where all the dtbo are cleared from the system.

**clear_dict**()

Clear all the dictionaries stored in PL.

This method will clear all the related dictionaries, including IP dictionary, GPIO dictionary, etc.

**close**()

**devicetree_dict**

The getter for the attribute *devicetree_dict*

> **Returns** The dictionary containing the device tree blobs.
>
> **Return type** dict

**get_bitfile_metadata**(*bitfile_name*)

**gpio_dict**

The getter for the attribute *gpio_dict*.

> **Returns** The dictionary storing the PS GPIO pins.
>
> **Return type** dict

**has_capability**(*cap*)

Test if the device as a desired capability

> **Parameters** **cap** (*str*) – The desired capability
>
> **Returns** True if the devices support cap
>
> **Return type** bool

**hierarchy_dict**

The getter for the attribute *hierarchy_dict*

> **Returns** The dictionary containing the hierarchies in the design
>
> **Return type** dict

**insert_device_tree**(*abs_dtbo*)

Insert device tree segment.

For device tree segments associated with full / partial bitstreams, users can provide the relative or absolute paths of the dtbo files.

> **Parameters** **abs_dtbo** (*str*) – The absolute path to the device tree segment.

**interrupt_controllers**
　　The getter for the attribute *interrupt_controllers*.

　　　　**Returns** The dictionary storing interrupt controller information.

　　　　**Return type** dict

**interrupt_pins**
　　The getter for the attribute *interrupt_pins*.

　　　　**Returns** The dictionary storing the interrupt endpoint information.

　　　　**Return type** dict

**ip_dict**
　　The getter for the attribute *ip_dict*.

　　　　**Returns** The dictionary storing addressable IP instances; can be empty.

　　　　**Return type** dict

**load_ip_data**(*ip_name*, *data*, *zero=False*)
　　This method writes data to the addressable IP.

---

**Note:** The data is assumed to be in binary format (.bin). The data name will be stored as a state information in the IP dictionary.

---

　　　　**Parameters**

　　　　　　• **ip_name** (*str*) – The name of the addressable IP.

　　　　　　• **data** (*str*) – The absolute path of the data to be loaded.

　　　　　　• **zero** (*bool*) – Zero out the address of the IP not covered by data

　　　　**Returns**

　　　　**Return type** None

**mem_dict**
　　The getter for the attribute *mem_dict*

　　　　**Returns** The dictionary containing the memory spaces in the design

　　　　**Return type** dict

**post_download**(*bitstream*, *parser*)

**remove_device_tree**(*abs_dtbo*)
　　Remove device tree segment for the overlay.

　　　　**Parameters** **abs_dtbo** (*str*) – The absolute path to the device tree segment.

**reset**(*parser=None*, *timestamp=None*, *bitfile_name=None*)
　　Reset all the dictionaries.

　　This method must be called after a bitstream download. 1. In case there is a *hwh* or *tcl* file, this method will reset the states of the IP, GPIO, and interrupt dictionaries . 2. In case there is no *hwh* or *tcl* file, this method will simply clear the state information stored for all dictionaries.

　　An existing parser given as the input can significantly reduce the reset time, since the PL can reset based on the information provided by the parser.

　　　　**Parameters**

- **parser** (*TCL/HWH*) – A parser object to speed up the reset process.
- **timestamp** (*str*) – The timestamp to embed in the reset
- **bitfile_name** (*str*) – The bitfile being loaded as part of the reset

**shutdown**()
> Shutdown the AXI connections to the PL in preparation for reconfiguration

**timestamp**
> The getter for the attribute *timestamp*.

> > **Returns** Bitstream download timestamp.

> > **Return type** str

**update_partial_region**(*hier*, *parser*)
> Merge the parser information from partial region.

> Combine the currently PL information and the partial HWH/TCL file parsing results.

> > **Parameters**

> > - **hier** (*str*) – The name of the hierarchical block as the partial region.
> > - **parser** (*TCL/HWH*) – A parser object for the partial region.

**class** pynq.pl_server.device.**DeviceMeta**(*name*, *bases*, *attrs*)
> Bases: `type`

> Metaclass for all types of Device

> It is responsible for enumerating the devices in the system and selecting a *default_device* that is used by applications that are oblivious to multiple-device scenarios

> The main implementation is the *Device* class which should be subclassed for each type of hardware that is supported. Each subclass should have a *_probe_* function which returns an array of *Device* objects and a *_probe_priority_* constant which is used to determine the default device.

> **active_device**
> > The device used by PYNQ if *None* used for a device parameter

> > This defaults to the device with the lowest priority and index but can be overridden to globally change the default.

> **devices**
> > All devices found in the system

> > An array of *Device* objects. Probing is done when this property is first accessed

**class** pynq.pl_server.device.**XlnkDevice**
> Bases: *pynq.pl_server.device.Device*

> Device sub-class for Xlnk based devices

> **BS_FPGA_MAN = '/sys/class/fpga_manager/fpga0/firmware'**

> **BS_FPGA_MAN_FLAGS = '/sys/class/fpga_manager/fpga0/flags'**

> **download**(*bitstream*, *parser=None*)

> **get_bitfile_metadata**(*bitfile_name*)

> **get_memory**(*description*)

> **mmap**(*base_addr*, *length*)

```
pynq.pl_server.device.parse_bit_header(bitfile)
```
   The method to parse the header of a bitstream.

   The returned dictionary has the following keys: "design": str, the Vivado project name that generated the bitstream; "version": str, the Vivado tool version that generated the bitstream; "part": str, the Xilinx part name that the bitstream targets; "date": str, the date the bitstream was compiled on; "time": str, the time the bitstream finished compilation; "length": int, total length of the bitstream (in bytes); "data": binary, binary data in .bit file format

   > **Returns** A dictionary containing the header information.

   > **Return type** Dict

   ---

   **Note:** Implemented based on: https://blog.aeste.my/?p=2892

   ---

## pynq.pl_server.xrt_device Module

The pynq.pl_server.xrt_device module extends the device class to work with Xilinx Run Time (XRT) enabled devices.

**class** `pynq.pl_server.xrt_device.`**`XrtDevice`**(*index*)
   Bases: `pynq.pl_server.device.Device`

   **`allocate_bo`**(*size*, *idx*)

   **`close`**()

   **`download`**(*bitstream*, *parser=None*)

   **`flush`**(*bo*, *offset*, *ptr*, *size*)

   **`get_bitfile_metadata`**(*bitfile_name*)

   **`get_device_address`**(*bo*)

   **`get_memory`**(*desc*)

   **`invalidate`**(*bo*, *offset*, *ptr*, *size*)

   **`map_bo`**(*bo*)

   **`name`**

   **`read_registers`**(*address*, *length*)

   **`write_registers`**(*address*, *data*)

**class** `pynq.pl_server.xrt_device.`**`XrtMemory`**(*device*, *desc*)
   Bases: `object`

   **`allocate`**(*shape*, *dtype*)

**class** `pynq.pl_server.xrt_device.`**`XrtUUID`**(*val*)
   Bases: `object`

## pynq.pl_server.hwh_parser Module

The pynq.pl_server.hwh_parser module parses the metadata file (hwh file). The goal is to extract useful information about the bitstream, including IP parameters, system clocks, block hierarchy, interruputs, and many others. This is the recommended way of getting overlay information.

pynq.pl_server.hwh_parser.**HWH**
    alias of pynq.pl_server.hwh_parser._HWHABC

pynq.pl_server.hwh_parser.**get_hwh_name**(*bitfile_name*)
    This method returns the name of the hwh file.

    For example, the input "/home/xilinx/pynq/overlays/base/base.bit" will lead to the result "/home/xilinx/pynq/overlays/base/base.hwh".

    > **Parameters bitfile_name** (*str*) – The absolute path of the .bit file.

    > **Returns** The absolute path of the .hwh file.

    > **Return type** str

pynq.pl_server.hwh_parser.**string2int**(*a*)
    Convert a hex or decimal string into an int.

    > **Parameters a** (*string*) – The input string representation of the number.

    > **Returns** The decimal number.

    > **Return type** int

## pynq.pl_server.tcl_parser Module

The pynq.pl_server.tcl_parser module parses the metadata file (tcl file). The goal is to extract useful information about the bitstream, including IP parameters, system clocks, block hierarchy, interruputs, and many others. The use of tcl files as a metadata file gets obsolete; it is to be retired in the future.

pynq.pl_server.tcl_parser.**TCL**
    alias of pynq.pl_server.tcl_parser._TCLABC

pynq.pl_server.tcl_parser.**get_tcl_name**(*bitfile_name*)
    This method returns the name of the tcl file.

    For example, the input "/home/xilinx/pynq/overlays/base/base.bit" will lead to the result "/home/xilinx/pynq/overlays/base/base.tcl".

    > **Parameters bitfile_name** (*str*) – The absolute path of the .bit file.

    > **Returns** The absolute path of the .tcl file.

    > **Return type** str

## pynq.pl_server.xclbin_parser Module

The pynq.pl_server.xclbin_parser module parses the metadata file from the xclbin file. It extracts useful information about the system such as the memory ports.

**class** pynq.pl_server.xclbin_parser.**XclBin**(*filename*)
    Bases: object

    Helper Class to extract information from an xclbin file

---

**Note:** This class requires the absolute path of the '.xclbin' file. Most of the dictionaries are empty to ensure compatibility with the HWH and TCL files.

---

**ip_dict**

> All the addressable IPs from PS7. Key is the name of the IP; value is a dictionary mapping the physical address, address range, IP type, memory segment ID, the state associated with that IP, any interrupts and GPIO pins attached to the IP and the full path to the IP in the block design: {str: {'phys_addr' : int, 'addr_range' : int, 'type' : str, 'mem_id' : str, 'state' : str, 'interrupts' : dict, 'gpio' : dict, 'fullpath' : str}}.
>
> > **Type** dict

**mem_dict**

> All of the memory regions and streaming connections in the design: {str: {'used' : bool, 'base_address' : int, 'size' : int, 'idx' : int, 'raw_type' : int, 'tyoe' : str, 'streaming' : bool}}.
>
> > **Type** dict

### 2.8.3 pynq.bitstream Module

The pynq.pl module facilitates the bitstream class to be used on the Programmable Logic (PL). The Bitstream class manages downloading of bitstreams onto the PL. It converts the bit files into bin files and leverages the FPGA manager framework to download them onto the PL. It handles full bitstreams and partial bitstreams. It also manages the device tree segment associated with each full or partial bitstream.

**class** pynq.bitstream.**Bitstream**(*bitfile_name*, *dtbo=None*, *partial=False*, *device=None*)

> Bases: `object`
>
> This class instantiates the meta class for PL bitstream (full/partial).
>
> **bitfile_name**
>
> > The absolute path or name of the bit file as a string.
> >
> > > **Type** str
>
> **dtbo**
>
> > The absolute path of the dtbo file as a string.
> >
> > > **Type** str
>
> **partial**
>
> > Flag to indicate whether or not the bitstream is partial.
> >
> > > **Type** bool
>
> **bit_data**
>
> > Dictionary storing information about the bitstream.
> >
> > > **Type** dict
>
> **binfile_name**
>
> > The absolute path or name of the bin file as a string.
> >
> > > **Type** str
>
> **firmware_path**
>
> > The absolute path of the bin file in the firmware folder.
> >
> > > **Type** str
>
> **timestamp**
>
> > Timestamp when loading the bitstream. Format: year, month, day, hour, minute, second, microsecond
> >
> > > **Type** str

**download**(*parser=None*)

>   Download the bitstream onto PL and update PL information.

>   If device tree blob has been specified during initialization, this method will also insert the corresponding device tree blob into the system. This is same for both full bitstream and partial bitstream.

---

>   **Note:** For partial bitstream, this method does not guarantee isolation between static and dynamic regions.

---

>   **Returns**

>   **Return type** None

**insert_dtbo**(*dtbo=None*)

>   Insert dtbo file into the system.

>   A simple wrapper of the corresponding method in the PL class. If *dtbo* is None, *self.dtbo* will be used to insert the dtbo file. In most cases, users should just ignore the parameter *dtbo*.

>   > **Parameters dtbo** (`str`) – The relative or absolute path to the device tree segment.

**remove_dtbo**()

>   Remove dtbo file from the system.

>   A simple wrapper of the corresponding method in the PL class. This is very useful for partial bitstream downloading, where loading the new device tree blob will overwrites the existing device tree blob in the same partial region.

## 2.8.4 pynq.buffer Module

Home of the `pynq.allocate` function

**class** `pynq.buffer.`**PynqBuffer**

>   Bases: `numpy.ndarray`

>   A subclass of numpy.ndarray which is allocated using physically contiguous memory for use with DMA engines and hardware accelerators. As physically contiguous memory is a limited resource it is strongly recommended to free the underlying buffer with *close* when the buffer is no longer needed. Alternatively a *with* statement can be used to automatically free the memory at the end of the scope.

>   This class should not be constructed directly and instead created using *pynq.allocate()*.

>   **device_address**

>   >   The physical address to the array

>   >   > **Type** int

>   **coherent**

>   >   Whether the buffer is coherent

>   >   > **Type** bool

>   **cacheable**

>   **close**()

>   >   Unused - for backwards compatibility only

>   **flush**()

>   >   Flush the underlying memory if necessary

---

**freebuffer**()
>   Free the underlying memory

>   This will free the memory regardless of whether other objects may still be using the buffer so ensure that no other references to the array exist prior to freeing.

**invalidate**()
>   Invalidate the underlying memory if necessary

**physical_address**

pynq.buffer.**allocate**(*shape*, *dtype*, *target=None*, *\*\*kwargs*)
>   Allocate a PYNQ buffer

>   The target determines where the buffer gets allocated

>   - If None then the currently active device is used

>   - If a Device is specified then the main memory

## 2.8.5 pynq.devicetree Module

The pynq.devicetree module enables users to insert or remove a device tree segment. This can happen when users load a full/partial bitstream.

**class** pynq.devicetree.**DeviceTreeSegment**(*dtbo_path*)
>   Bases: `object`

>   This class instantiates the device tree segment object.

>   **dtbo_name**
>>   The base name of the dtbo file as a string.

>>>   **Type** str

>   **dtbo_path**
>>   The absolute path to the dtbo file as a string.

>>>   **Type** str

>   **insert**()
>>   Insert the dtbo file into the device tree.

>>   The method will raise an exception if the insertion has failed.

>   **is_dtbo_applied**()
>>   Show if the device tree segment has been applied.

>>>   **Returns** True if the device tree status shows *applied*.

>>>   **Return type** bool

>   **remove**()
>>   Remove the dtbo file from the device tree.

pynq.devicetree.**get_dtbo_base_name**(*dtbo_path*)
>   This method returns the base name of the dtbo file.

>   For example, the input "/home/xilinx/pynq/overlays/name1/name2.dtbo" will lead to the result "name2".

>   **Parameters** **dtbo_path** (`str`) – The absolute path of the dtbo file.

>   **Returns** The base name of the dtbo file.

>   **Return type** str

pynq.devicetree.**get_dtbo_path**(*bitfile_name*)

> This method returns the path of the dtbo file.
>
> For example, the input "/home/xilinx/pynq/overlays/base/base.bit" will lead to the result "/home/xilinx/pynq/overlays/base/base.dtbo".
>
> > **Parameters** **bitfile_name** (`str`) – The absolute path of the bit file.
> >
> > **Returns** The absolute path of the dtbo file.
> >
> > **Return type** str

## 2.8.6 pynq.gpio Module

The pynq.gpio module is a driver for reading and writing PS GPIO pins on a board. PS GPIO pins are not connected to the PL.

**class** pynq.gpio.**GPIO**(*gpio_index*, *direction*)

> Bases: `object`
>
> Class to wrap Linux's GPIO Sysfs API.
>
> This GPIO class does not handle PL I/O without the use of device tree overlays.
>
> **index**
>
> > The index of the GPIO, starting from the GPIO base.
> >
> > > **Type** int
>
> **direction**
>
> > Input/output direction of the GPIO.
> >
> > > **Type** str
>
> **path**
>
> > The path of the GPIO device in the linux system.
> >
> > > **Type** str
>
> **direction**
>
> **static get_gpio_base**(*target_label=None*)
>
> > This method returns the GPIO base using Linux's GPIO Sysfs API.
> >
> > This is a static method. To use:
> >
> > ```
> > >>> from pynq import GPIO
> > ```
> >
> > ```
> > >>> gpio = GPIO.get_gpio_base()
> > ```
> >
> > ---
> >
> > **Note:** For path '/sys/class/gpio/gpiochip138/', this method returns 138.
> >
> > ---
> >
> > > **Parameters** **target_label** (`str`) – The label of the GPIO driver to look for, as defined in a device tree entry.
> > >
> > > **Returns** The GPIO index of the base.
> > >
> > > **Return type** int

static **get_gpio_base_path**(*target_label=None*)

This method returns the path to the GPIO base using Linux's GPIO Sysfs API.

This is a static method. To use:

```
>>> from pynq import GPIO
```

```
>>> gpio = GPIO.get_gpio_base_path()
```

> **Parameters** **target_label** (`str`) – The label of the GPIO driver to look for, as defined in a device tree entry.
>
> **Returns** The path to the GPIO base.
>
> **Return type** str

static **get_gpio_npins**(*target_label=None*)

This method returns the number of GPIO pins for the GPIO base using Linux's GPIO Sysfs API.

This is a static method. To use:

```
>>> from pynq import GPIO
```

```
>>> gpio = GPIO.get_gpio_npins()
```

> **Parameters** **target_label** (`str`) – The label of the GPIO driver to look for, as defined in a device tree entry.
>
> **Returns** The number of GPIO pins for the GPIO base.
>
> **Return type** int

static **get_gpio_pin**(*gpio_user_index*, *target_label=None*)

This method returns a GPIO instance for PS GPIO pins.

Users only need to specify an index starting from 0; this static method will map this index to the correct Linux GPIO pin number.

---

**Note:** The GPIO pin number can be calculated using: GPIO pin number = GPIO base + GPIO offset + user index e.g. The GPIO base is 138, and pin 54 is the base GPIO offset. Then the Linux GPIO pin would be (138 + 54 + 0) = 192.

---

> **Parameters**
>
> - **gpio_user_index** (`int`) – The index specified by users, starting from 0.
> - **target_label** (`str`) – The label of the GPIO driver to look for, as defined in a device tree entry.
>
> **Returns** The Linux Sysfs GPIO pin number.
>
> **Return type** int

**index**

**path**

---

**read**()
>    The method to read a value from the GPIO.

>    >    **Returns** An integer read from the GPIO

>    >    **Return type** int

**release**()
>    The method to release the GPIO.

>    >    **Returns**

>    >    **Return type** None

**write**(*value*)
>    The method to write a value into the GPIO.

>    >    **Parameters** **value** (*int*) – An integer value, either 0 or 1

>    >    **Returns**

>    >    **Return type** None

## 2.8.7  pynq.interrupt Module

**class** pynq.interrupt.**Interrupt**(*pinname*)
>    Bases: `object`

>    Class that provides the core wait-based API to end users

>    Provides a single coroutine wait that waits until the interrupt signal goes high.  If the Overlay is changed or re-downloaded this object is invalidated and waiting results in undefined behaviour.

>    **wait**()
>    >    Wait for the interrupt to be active

>    >    May raise an exception if the Overlay has been changed since initialisation.

pynq.interrupt.**get_uio_irq**(*irq*)
>    Returns the UIO device path for a specified interrupt.

>    If the IRQ either cannot be found or does not correspond to a UIO device, None is returned.

>    >    **Parameters** **irq** (*int*) – The desired physical interrupt line

>    >    **Returns** The path of the device in /dev list.

>    >    **Return type** str

## 2.8.8  pynq.mmio Module

**class** pynq.mmio.**MMIO**(*base_addr*, *length=4*, *debug=False*, *device=None*)
>    Bases: `object`

>    This class exposes API for MMIO read and write.

>    **base_addr**
>    >    The base address, not necessarily page aligned.

>    >    **Type** int

>    **length**
>    >    The length in bytes of the address range.

**Type** int

**debug**
    Turn on debug mode if it is True.

        **Type** bool

**array**
    A numpy view of the mapped range for efficient assignment

        **Type** numpy.ndarray

**read_mm**(*offset=0*, *length=4*)
    The method to read data from MMIO.

        **Parameters**

            • **offset** (*int*) – The read offset from the MMIO base address.

            • **length** (*int*) – The length of the data in bytes.

        **Returns** A list of data read out from MMIO

        **Return type** list

**read_reg**(*offset=0*, *length=4*)
    The method to read data from MMIO.

        **Parameters**

            • **offset** (*int*) – The read offset from the MMIO base address.

            • **length** (*int*) – The length of the data in bytes.

        **Returns** A list of data read out from MMIO

        **Return type** list

**write_mm**(*offset*, *data*)
    The method to write data to MMIO.

        **Parameters**

            • **offset** (*int*) – The write offset from the MMIO base address.

            • **data** (*int / bytes*) – The integer(s) to be written into MMIO.

        **Returns**

        **Return type** None

**write_reg**(*offset*, *data*)
    The method to write data to MMIO.

        **Parameters**

            • **offset** (*int*) – The write offset from the MMIO base address.

            • **data** (*int / bytes*) – The integer(s) to be written into MMIO.

        **Returns**

        **Return type** None

## 2.8.9 pynq.overlay Module

The pynq.overlay module inherits from the PL module and is used to manage the state and contents of a PYNQ Overlay. The module adds additional functionality to the PL module. For example, the PL module contains the methods to download the overlay file. The Overlay module sets the PL clocks and ARM architecture registers before before calling the Bitstream download() method.

**class** `pynq.overlay.`**`DefaultHierarchy`**(*description*)

  Bases: `pynq.overlay._IPMap`

  Hierarchy exposing all IP and hierarchies as attributes

  This Hierarchy is instantiated if no more specific hierarchy class registered with register_hierarchy_driver is specified. More specific drivers should inherit from *DefaultHierarachy* and call it's constructor in __init__ prior to any other initialisation. *checkhierarchy* should also be redefined to return True if the driver matches a hierarchy. Any derived class that meets these requirements will automatically be registered in the driver database.

  **description**

    Dictionary storing relevant information about the hierarchy.

      **Type** dict

  **parsers**

    Parser objects for partial block design metadata.

      **Type** dict

  **bitstreams**

    Bitstream objects for partial designs.

      **Type** dict

  **pr_loaded**

    The absolute path of the partial bitstream loaded.

      **Type** str

  **static checkhierarchy**(*description*)

    Function to check if the driver matches a particular hierarchy

    This function should be redefined in derived classes to return True if the description matches what is expected by the driver. The default implementation always returns False so that drivers that forget don't get loaded for hierarchies they don't expect.

  **download**(*bitfile_name*, *dtbo*)

    Function to download a partial bitstream for the hierarchy block.

    Since it is hard to know which hierarchy is to be reconfigured by only looking at the metadata, we assume users will tell this information. Thus, this function should be called only when users are sure about the hierarchy name of the partial region.

      **Parameters**

        • **bitfile_name** (*str*) – The name of the partial bitstream.

        • **dtbo** (*str*) – The relative or absolute path of the partial dtbo file.

**class** `pynq.overlay.`**`DefaultIP`**(*description*)

  Bases: `object`

  Driver for an IP without a more specific driver

This driver wraps an MMIO device and provides a base class for more specific drivers written later. It also provides access to GPIO outputs and interrupts inputs via attributes. More specific drivers should inherit from *DefaultIP* and include a *bindto* entry containing all of the IP that the driver should bind to. Subclasses meeting these requirements will automatically be registered.

**mmio**
>    Underlying MMIO driver for the device
>
>    **Type**  pynq.MMIO

**_interrupts**
>    Subset of the PL.interrupt_pins related to this IP
>
>    **Type**  dict

**_gpio**
>    Subset of the PL.gpio_dict related to this IP
>
>    **Type**  dict

**call**(*\*args*, *\*\*kwargs*)

**read**(*offset=0*)
>    Read from the MMIO device
>
>    **Parameters offset** (`int`) – Address to read

**register_map**

**signature**
>    The signature of the *call* method

**start**(*\*args*, *\*\*kwargs*)
>    Start the accelerator
>
>    This function will configure the accelerator with the provided arguments and start the accelerator. Use the *wait* function to determine when execution has finished. Note that buffers should be flushed prior to starting the accelerator and any result buffers will need to be invalidated afterwards.
>
>    For details on the function's signature use the *signature* property. The type annotations provide the C types that the accelerator operates on. Any pointer types should be passed as *ContiguousArray* objects created from the *pynq.Xlnk* class. Scalars should be passed as a compatible python type as used by the *struct* library.

**start_sw**(*\*args*, *ap_ctrl=1*, *\*\*kwargs*)
>    Start the accelerator
>
>    This function will configure the accelerator with the provided arguments and start the accelerator. Use the *wait* function to determine when execution has finished. Note that buffers should be flushed prior to starting the accelerator and any result buffers will need to be invalidated afterwards.
>
>    For details on the function's signature use the *signature* property. The type annotations provide the C types that the accelerator operates on. Any pointer types should be passed as *ContiguousArray* objects created from the *pynq.Xlnk* class. Scalars should be passed as a compatible python type as used by the *struct* library.

**write**(*offset*, *value*)
>    Write to the MMIO device
>
>    **Parameters**
>
>    - **offset** (`int`) – Address to write to
>    - **value** (`int or bytes`) – Data to write

---

pynq.overlay.**DocumentHierarchy**(*description*)

    Helper function to build a custom hierarchy class with a docstring based on the description. Mimics a class constructor

pynq.overlay.**DocumentOverlay**(*bitfile*, *download*)

    Function to build a custom overlay class with a custom docstring based on the supplied bitstream. Mimics a class constructor.

**class** pynq.overlay.**Overlay**(*bitfile_name*, *dtbo=None*, *download=True*, *ignore_version=False*, *device=None*)

    Bases: `pynq.bitstream.Bitstream`

    This class keeps track of a single bitstream's state and contents.

    The overlay class holds the state of the bitstream and enables run-time protection of bindlings.

    Our definition of overlay is: "post-bitstream configurable design". Hence, this class must expose configurability through content discovery and runtime protection.

    The overlay class exposes the IP and hierarchies as attributes in the overlay. If no other drivers are available the *DefaultIP* is constructed for IP cores at top level and *DefaultHierarchy* for any hierarchies that contain addressable IP. Custom drivers can be bound to IP and hierarchies by subclassing *DefaultIP* and *DefaultHierarchy*. See the help entries for those class for more details.

    This class stores four dictionaries: IP, GPIO, interrupt controller and interrupt pin dictionaries.

    Each entry of the IP dictionary is a mapping: 'name' -> {phys_addr, addr_range, type, config, state}, where name (str) is the key of the entry. phys_addr (int) is the physical address of the IP. addr_range (int) is the address range of the IP. type (str) is the type of the IP. config (dict) is a dictionary of the configuration parameters. state (str) is the state information about the IP.

    Each entry of the GPIO dictionary is a mapping: 'name' -> {pin, state}, where name (str) is the key of the entry. pin (int) is the user index of the GPIO, starting from 0. state (str) is the state information about the GPIO.

    Each entry in the interrupt controller dictionary is a mapping: 'name' -> {parent, index}, where name (str) is the name of the interrupt controller. parent (str) is the name of the parent controller or '' if attached directly to the PS. index (int) is the index of the interrupt attached to.

    Each entry in the interrupt pin dictionary is a mapping: 'name' -> {controller, index}, where name (str) is the name of the pin. controller (str) is the name of the interrupt controller. index (int) is the line index.

    **bitfile_name**

        The absolute path of the bitstream.

            **Type**   str

    **dtbo**

        The absolute path of the dtbo file for the full bitstream.

            **Type**   str

    **ip_dict**

        All the addressable IPs from PS. Key is the name of the IP; value is a dictionary mapping the physical address, address range, IP type, parameters, registers, and the state associated with that IP: {str: {'phys_addr' : int, 'addr_range' : int, 'type' : str, 'parameters' : dict, 'registers': dict, 'state' : str}}.

            **Type**   dict

    **gpio_dict**

        All the GPIO pins controlled by PS. Key is the name of the GPIO pin; value is a dictionary mapping user index (starting from 0), and the state associated with that GPIO pin: {str: {'index' : int, 'state' : str}}.

            **Type**   dict

**interrupt_controllers**
> All AXI interrupt controllers in the system attached to a PS interrupt line. Key is the name of the controller; value is a dictionary mapping parent interrupt controller and the line index of this interrupt: {str: {'parent': str, 'index' : int}}. The PS is the root of the hierarchy and is unnamed.
>
> > **Type** dict

**interrupt_pins**
> All pins in the design attached to an interrupt controller. Key is the name of the pin; value is a dictionary mapping the interrupt controller and the line index used: {str: {'controller' : str, 'index' : int}}.
>
> > **Type** dict

**pr_dict**
> Dictionary mapping from the name of the partial-reconfigurable hierarchical blocks to the loaded partial bitstreams: {str: {'loaded': str, 'dtbo': str}}.
>
> > **Type** dict

**download**(*dtbo=None*)
> The method to download a full bitstream onto PL.
>
> After the bitstream has been downloaded, the "timestamp" in PL will be updated. In addition, all the dictionaries on PL will be reset automatically.
>
> This method will use parameter *dtbo* or *self.dtbo* to configure the device tree.
>
> > **Parameters dtbo** (*str*) – The path of the dtbo file.

**is_loaded**()
> This method checks whether a bitstream is loaded.
>
> This method returns true if the loaded PL bitstream is same as this Overlay's member bitstream.
>
> > **Returns** True if bitstream is loaded.
>
> > **Return type** bool

**load_ip_data**(*ip_name*, *data*)
> This method loads the data to the addressable IP.
>
> Calls the method in the super class to load the data. This method can be used to program the IP. For example, users can use this method to load the program to the Microblaze processors on PL.

---

**Note:** The data is assumed to be in binary format (.bin). The data name will be stored as a state information in the IP dictionary.

---

> > **Parameters**
> >
> > - **ip_name** (*str*) – The name of the addressable IP.
> >
> > - **data** (*str*) – The absolute path of the data to be loaded.
> >
> > **Returns**
> >
> > **Return type** None

**pr_download**(*partial_region*, *partial_bit*, *dtbo=None*)
> The method to download a partial bitstream onto PL.
>
> In this method, the corresponding parser will only be added once the *download()* method of the hierarchical block is called.

This method always uses the parameter *dtbo* to configure the device tree.

---

**Note:** There is no check on whether the partial region specified by users is really partial-reconfigurable. So users have to make sure the *partial_region* provided is correct.

---

> Parameters
>> • **partial_region** (*str*) – The name of the hierarchical block corresponding to the PR region.
>>
>> • **partial_bit** (*str*) – The name of the partial bitstream.
>>
>> • **dtbo** (*str*) – The path of the dtbo file.

**reset**()
> This function resets all the dictionaries kept in the overlay.
>
> This function should be used with caution. In most cases, only those dictionaries keeping track of states need to be updated.
>> Returns
>> Return type None

**class** pynq.overlay.**RegisterHierarchy**(*name*, *bases*, *attrs*)
> Bases: type

Metaclass to register classes as hierarchy drivers

Any class with this metaclass an the *checkhierarchy* function will be registered in the global driver database

**class** pynq.overlay.**RegisterIP**(*name*, *bases*, *attrs*)
> Bases: type

Meta class that binds all registers all subclasses as IP drivers

The *bindto* attribute of subclasses should be an array of strings containing the VLNV of the IP the driver should bind to.

**class** pynq.overlay.**WaitHandle**(*target*)
> Bases: object

**wait**()

**class** pynq.overlay.**XrtArgument**(*name*, *index*, *type*, *mem*)
> Bases: tuple

**index**
> Alias for field number 1

**mem**
> Alias for field number 3

**name**
> Alias for field number 0

**type**
> Alias for field number 2

### 2.8.10 pynq.pl Module

The pynq.pl module facilitates management of the Programmable Logic (PL). The PL module manages the PL state through the PL class. The PL class is a singleton for the Overlay class and Bitstream classes that provide user-facing methods for bitstream and overlay manipulation.

**class** `pynq.pl.`**PL**
Bases: `object`

Serves as a singleton for *Overlay* and *Bitstream* classes.

This class stores multiple dictionaries: IP dictionary, GPIO dictionary, interrupt controller dictionary, and interrupt pins dictionary.

**bitfile_name**
The absolute path of the bitstream currently on PL.

> **Type** str

**timestamp**
Bitstream download timestamp, using the following format: year, month, day, hour, minute, second, microsecond.

> **Type** str

**ip_dict**
All the addressable IPs from PS7. Key is the name of the IP; value is a dictionary mapping the physical address, address range, IP type, configuration dictionary, the state associated with that IP, any interrupts and GPIO pins attached to the IP and the full path to the IP in the block design: {str: {'phys_addr' : int, 'addr_range' : int, 'type' : str, 'config' : dict, 'state' : str, 'interrupts' : dict, 'gpio' : dict, 'fullpath' : str}}.

> **Type** dict

**gpio_dict**
All the GPIO pins controlled by PS7. Key is the name of the GPIO pin; value is a dictionary mapping user index (starting from 0), the state associated with that GPIO pin and the pins in block diagram attached to the GPIO: {str: {'index' : int, 'state' : str, 'pins' : [str]}}.

> **Type** dict

**interrupt_controllers**
All AXI interrupt controllers in the system attached to a PS7 interrupt line. Key is the name of the controller; value is a dictionary mapping parent interrupt controller and the line index of this interrupt: {str: {'parent': str, 'index' : int}}. The PS7 is the root of the hierarchy and is unnamed.

> **Type** dict

**interrupt_pins**
All pins in the design attached to an interrupt controller. Key is the name of the pin; value is a dictionary mapping the interrupt controller and the line index used: {str: {'controller' : str, 'index' : int}}.

> **Type** dict

**hierarchy_dict**
All of the hierarchies in the block design containing addressable IP. The keys are the hiearachies and the values are dictionaries containing the IP and sub-hierarchies contained in the hierarchy and and GPIO and interrupts attached to the hierarchy. The keys in dictionaries are relative to the hierarchy and the ip dict only contains immediately contained IP - not those in sub-hierarchies. {str: {'ip': dict, 'hierarchies': dict, 'interrupts': dict, 'gpio': dict, 'fullpath': str}}

> **Type** dict

**class** pynq.pl.**PLMeta**

Bases: type

This method is the meta class for the PL.

This is not a class for users. Hence there is no attribute or method exposed to users.

We make no assumption of the overlay during boot, so most of the dictionaries are empty. Those dictionaries will get populated when users download an overlay onto the PL.

---

**Note:** If this metaclass is parsed on an unsupported architecture it will issue a warning and leave class variables undefined

---

**bitfile_name**

The getter for the attribute *bitfile_name*.

> **Returns** The absolute path of the bitstream currently on PL.
>
> **Return type** str

**clear_devicetree**()

Clear the device tree dictionary.

This should be used when downloading the full bitstream, where all the dtbo are cleared from the system.

**clear_dict**()

Clear all the dictionaries stored in PL.

This method will clear all the related dictionaries, including IP dictionary, GPIO dictionary, etc.

**devicetree_dict**

The getter for the attribute *devicetree_dict*

> **Returns** The dictionary containing the device tree blobs.
>
> **Return type** dict

**gpio_dict**

The getter for the attribute *gpio_dict*.

> **Returns** The dictionary storing the PS GPIO pins.
>
> **Return type** dict

**hierarchy_dict**

The getter for the attribute *hierarchy_dict*

> **Returns** The dictionary containing the hierarchies in the design
>
> **Return type** dict

**insert_device_tree**(*abs_dtbo*)

Insert device tree segment.

For device tree segments associated with full / partial bitstreams, users can provide the relative or absolute paths of the dtbo files.

> **Parameters** **abs_dtbo** (`str`) – The absolute path to the device tree segment.

**interrupt_controllers**

The getter for the attribute *interrupt_controllers*.

> **Returns** The dictionary storing interrupt controller information.
>
> **Return type** dict

---

**interrupt_pins**
>    The getter for the attribute *interrupt_pins*.

>>        **Returns** The dictionary storing the interrupt endpoint information.

>>        **Return type** dict

**ip_dict**
>    The getter for the attribute *ip_dict*.

>>        **Returns** The dictionary storing addressable IP instances; can be empty.

>>        **Return type** dict

**load_ip_data**(*ip_name*, *data*, *zero=False*)
>    This method writes data to the addressable IP.

---

**Note:** The data is assumed to be in binary format (.bin). The data name will be stored as a state information in the IP dictionary.

---

>        **Parameters**

>>            • **ip_name** (*str*) – The name of the addressable IP.

>>            • **data** (*str*) – The absolute path of the data to be loaded.

>>            • **zero** (*bool*) – Zero out the address of the IP not covered by data

>        **Returns**

>        **Return type** None

**mem_dict**
>    The getter for the attribute *mem_dict*

>>        **Returns** The dictionary containing the memories in the design.

>>        **Return type** dict

**remove_device_tree**(*abs_dtbo*)
>    Remove device tree segment for the overlay.

>>        **Parameters** **abs_dtbo** (*str*) – The absolute path to the device tree segment.

**reset**(*parser=None*)
>    Reset all the dictionaries.

>    This method must be called after a bitstream download. 1. In case there is a *hwh* or *tcl* file, this method will reset the states of the IP, GPIO, and interrupt dictionaries . 2. In case there is no *hwh* or *tcl* file, this method will simply clear the state information stored for all dictionaries.

>    An existing parser given as the input can significantly reduce the reset time, since the PL can reset based on the information provided by the parser.

>>        **Parameters** **parser** (*TCL/HWH*) – A parser object to speed up the reset process.

**shutdown**()
>    Shutdown the AXI connections to the PL in preparation for reconfiguration

**timestamp**
>    The getter for the attribute *timestamp*.

>>        **Returns** Bitstream download timestamp.

---

> **Return type** str

**update_partial_region**(*hier*, *parser*)

> Merge the parser information from partial region.
>
> Combine the currently PL information and the partial HWH/TCL file parsing results.
>
> > **Parameters**
> >
> > - **hier** (`str`) – The name of the hierarchical block as the partial region.
> >
> > - **parser** (`TCL/HWH`) – A parser object for the partial region.

## 2.8.11 pynq.pmbus Module

**class** pynq.pmbus.**DataRecorder**(*\*sensors*)

> Bases: `object`
>
> Class to record sensors during an execution
>
> The DataRecorder provides a way of recording sensor data using a *with* block.
>
> **frame**
>
> > Return a pandas DataFrame of the recorded data
> >
> > The frame consists of the following fields Index : The timestamp of the measurement Mark : counts the number of times that record or mark was called Sensors* : one column per sensor
>
> **mark**()
>
> > Increment the Invocation count
>
> **record**(*interval*)
>
> > Start recording
>
> **reset**()
>
> > Clear the internal state of the data recorder without forgetting which sensors to record
>
> **stop**()
>
> > Stops recording

**class** pynq.pmbus.**Rail**(*name*)

> Bases: `object`
>
> Bundles up to three sensors monitoring the same power rail
>
> Represents a power rail in the system monitored by up to three sensors for voltage, current and power.
>
> **name**
>
> > Name of the power rail
> >
> > > **Type** str
>
> **voltage**
>
> > Voltage sensor for the rail or None if not available
> >
> > > **Type** *Sensor* or None
>
> **current**
>
> > Current sensor for the rail or None if not available
> >
> > > **Type** *Sensor* or None
>
> **power**
>
> > Power sensor for the rail or None if not available

> **Type** *Sensor* or None

**class** `pynq.pmbus.`**`Sensor`**(*chip*, *number*, *unit*, *name*)

> Bases: `object`

> Interacts with a sensor exposed by libsensors

> The value of the sensor is determined by the unit of the underlying sensor API - that is generally Volts for potential difference, Amperes for current, Watts for power and degrees Centigrade for temperature

> **`name`**

>> The name of the sensor

>>> **Type** str

> **`value`**

>> The current value of the sensor

>>> **Type** float

> **`value`**

>> Read the current value of the sensor

`pynq.pmbus.`**`get_rails`**(*config_file=None*)

> Returns a dictionary of power rails

>> **Parameters** **`config_file`** (`str`) – Path to a configuration file for libsensors to use in place of the the system-wide default

>> **Returns** **dict {str** – Dictionary of power rails with the name of the rail as the key and a Rail object as the value

>> **Return type** Rail}

## 2.8.12 pynq.ps Module

The pynq.ps module facilitates management of the Processing System (PS) and PS/PL interface. It provides Clocks classes for setting and getting of Programmable Logic (PL) clocks.

**class** `pynq.ps.`**`Clocks`**

> Bases: `object`

> Class for all the PS and PL clocks exposed to users.

> With this class, users can get the CPU clock and all the PL clocks. Users can also set PL clocks to other values using this class.

> **`cpu_mhz`**

>> The clock rate of the CPU, measured in MHz.

>>> **Type** float

> **`fclk0_mhz`**

>> The clock rate of the PL clock 0, measured in MHz.

>>> **Type** float

> **`fclk1_mhz`**

>> The clock rate of the PL clock 1, measured in MHz.

>>> **Type** float

> **`fclk2_mhz`**

>> The clock rate of the PL clock 2, measured in MHz.

> **Type** float

**fclk3_mhz**

> The clock rate of the PL clock 3, measured in MHz.
>
> > **Type** float

## 2.8.13 pynq.registers Module

The pynq.registers module provides Register class for setting and getting of ARM Architecture register bits. This can be used for multiple purposes; for example, the Register class can be used for getting and setting the frequencies of Programmable Logic (PL) clocks.

**class** pynq.registers.**Register**(*address*, *width=32*, *debug=False*, *buffer=None*)

> Bases: object
>
> Register class that allows users to access registers easily.
>
> This class supports register slicing, which makes the access to register values much more easily. Users can either use +1 or -1 as the step when slicing the register. By default, the slice starts from MSB to LSB, which is consistent with the common hardware design practice.
>
> For example, the following slices are acceptable: reg[31:13] (commonly used), reg[:], reg[3::], reg[:20:], reg[1:3], etc.
>
> ---
>
> **Note:** The slicing endpoints are closed, meaning both of the 2 endpoints will be included in the final returned value. For example, reg[31:0] will return a 32-bit value; this is consistent with most of the hardware definitions.
>
> ---
>
> **address**
>
> > The address of the register.
> >
> > > **Type** int
>
> **width**
>
> > The width of the register, e.g., 32 (default) or 64.
> >
> > > **Type** int
>
> **classmethod count**(*index*, *width=32*)
>
> > Provide the number of bits accessed by an index or slice
> >
> > This method accepts both integer index, or slice as input parameters.
> >
> > > **Parameters**
> > >
> > > - **index** (*int | slice*) – The integer index, or slice to access the register value.
> > >
> > > - **width** (*int*) – The number of bits accessed.
>
> **classmethod create_subclass**(*name*, *fields*, *doc=None*)
>
> > Create a subclass of Register that has properties for the specified fields
> >
> > The fields should be in the form used by *ip_dict*, namely:
> >
> > ```
> > {name: {'access': "read-only" | "read-write" | "write-only",
> >         'bit_offset': int, 'bit_width': int, 'description': str}}
> > ```
> >
> > > **Parameters**
> > >
> > > - **name** (*str*) – A suffix for the name of the subclass

> • **fields** (*dict*) – A Dictionary containing the fields to add to the subclass

**class** pynq.registers.**RegisterMap**(*buffer*)

> Bases: object

> Provides access to a named register map.

> This class is designed to be subclassed using the *create_subclass* method which will create a class with properties for all of the registers in a specific map.

> See the *create_subclass* function for more details.

> **classmethod create_subclass**(*name*, *registers*)

> > Create a new RegisterMap subclass with the specified registers

> > The dictionary should have the same form as the "registers" entry in the ip_dict. For example:

> > ```
> > {name : {"address_offset" : int,
> >          "access" : "read-only" | "write-only" | "read-write",
> >          "size" : int,
> >          "description" : str,
> >          "fields" : dict}}
> > ```

> > For details on the contents of the "fields" entry see the *Register* class documentation.

> > **Parameters**

> > > • **name** (*str*) – Suffix to append to "RegisterMap" to make the name of the new class

> > > • **registers** (*dict*) – Dictionary of the registers to create in the subclass

## 2.8.14 pynq.uio Module

**class** pynq.uio.**UioController**(*device*)

> Bases: object

> Class that interacts directly with a UIO device.

> **uio**

> > File handle for the opened UIO.

> > > **Type** _io.BufferedRandom

> **add_event**(*event*, *number*)

pynq.uio.**get_uio_device**(*dev_name*)

> Returns the UIO device path.

> This method will return None if no such device can be found.

> > **Parameters dev_name** (*str*) – The name of the UIO device.

> > **Returns** The path of the device in /dev list.

> > **Return type** str

pynq.uio.**get_uio_index**(*name*)

> Return the uio index for the given device.

> > **Parameters name** (*str*) – The name of the UIO device.

> > **Returns** The index number for the UIO device.

> > **Return type** int

## 2.8.15 pynq.xlnk Module

This module is now deprecated in favour of `pynq.allocate`

**class** `pynq.xlnk.`**Xlnk**
> Bases: `object`
>
> Class to enable CMA memory management.
>
> The CMA state maintained by this class is local to the application except for the *CMA Memory Available* attribute which is global across all the applications.
>
> **bufmap**
>> Mapping of allocated memory to the buffer sizes in bytes.
>>
>>> **Type** dict
>
> **ffi**
>> Shared-object interface for the compiled CMA shared object
>>
>>> **Type** cffi instance
>
> ---
>
> **Note:** If this class is parsed on an unsupported architecture it will issue a warning and leave the class variable libxlnk undefined
>
> ---
>
> **allocate**(*\*args*, *\*\*kwargs*)
>> Wrapper for cma_array to match Xlnk to new Memory API
>
> **cma_alloc**(*length*, *cacheable=0*, *data_type='void'*)
>> Allocate physically contiguous memory buffer.
>>
>> Allocates a new buffer and adds it to *bufmap*.
>>
>> Possible values for parameter *cacheable* are:
>>
>> *1*: the memory buffer is cacheable.
>>
>> *0*: the memory buffer is non-cacheable.
>>
>> **Examples**
>>
>> mmu = Xlnk()
>>
>> # Allocate 10 *void* * memory locations.
>>
>> m1 = mmu.cma_alloc(10)
>>
>> # Allocate 10 *float* * memory locations.
>>
>> m2 = mmu.cma_alloc(10, data_type = "float")
>>
>> **Notes**
>>
>> 1. Total size of buffer is automatically calculated as size = length * sizeof(data_type)
>>
>> 2. This buffer is allocated inside the kernel space using xlnk driver. The maximum allocatable memory is defined at kernel build time using the CMA memory parameters.
>>
>> The unit of *length* depends upon the *data_type* argument.
>>
>>> **Parameters**

- **length** (*int*) – Length of the allocated buffer. Default unit is bytes.

- **cacheable** (*int*) – Indicating whether or not the memory buffer is cacheable.

- **data_type** (*str*) – CData type of the allocated buffer. Should be a valid C-Type.

**Returns** An CFFI object which can be accessed similar to arrays.

**Return type** cffi.FFI.CData

**cma_array** (*shape*, *dtype=<class 'numpy.uint32'>*, *cacheable=0*, *pointer=None*, *cache=None*)
Get a contiguously allocated numpy array

Create a numpy array with physically contiguously array. The physical address of the array can be found using the *physical_address* attribute of the returned object. The array should be freed using either *array.freebuffer()* or *array.close()* when the array is no longer required. Alternatively *cma_array* may be used in a *with* statement to automatically free the memory at the end of the block.

**Parameters**

- **shape** (*int or tuple of int*) – The dimensions of the array to construct

- **dtype** (*numpy.dtype or str*) – The data type to construct - defaults to 32-bit unsigned int

- **cacheable** (*int*) – Whether the buffer should be cacheable - defaults to 0

**Returns** The numpy array

**Return type** numpy.ndarray

**static cma_cast** (*data*, *data_type='void'*)
Cast underlying buffer to a specific C-Type.

Input buffer should be a valid object which was allocated through *cma_alloc* or a CFFI pointer to a memory buffer. Handy for changing void buffers to user defined buffers.

**Parameters**

- **data** (*cffi.FFI.CData*) – A valid buffer pointer allocated via *cma_alloc*.

- **data_type** (*str*) – New data type of the underlying buffer.

**Returns** Pointer to buffer with specified data type.

**Return type** cffi.FFI.CData

**cma_free** (*buf*)
Free a previously allocated buffer.

Input buffer should be a valid object which was allocated through *cma_alloc* or a CFFI pointer to a memory buffer.

**Parameters buf** (*cffi.FFI.CData*) – A valid buffer pointer allocated via *cma_alloc*.

**Returns**

**Return type** None

**cma_get_buffer** (*buf*, *length*)
Get a buffer object.

Used to get an object which supports python buffer interface. The return value thus, can be cast to objects like *bytearray*, *memoryview* etc.

**Parameters**

---

- **buf** (`cffi.FFI.CData`) – A valid buffer object which was allocated through *cma_alloc*.

- **length** (`int`) – Length of buffer in Bytes.

> **Returns** A CFFI object which supports buffer interface.

> **Return type** cffi.FFI.CData

**cma_get_phy_addr**(*buf_ptr*)
    Get the physical address of a buffer.

    Used to get the physical address of a memory buffer allocated with *cma_alloc*. The return value can be used to access the buffer from the programmable logic.

> **Parameters** **buf_ptr** (`cffi.FFI.CData`) – A void pointer pointing to the memory buffer.

> **Returns** The physical address of the memory buffer.

> **Return type** int

**cma_mem_size**()
    Get the total size of CMA memory in the system

> **Returns** The number of bytes of CMA memory

> **Return type** int

**static cma_memcopy**(*dest*, *src*, *nbytes*)
    High speed memcopy between buffers.

    Used to perform a byte level copy of data from source buffer to the destination buffer.

> **Parameters**

- **dest** (`cffi.FFI.CData`) – Destination buffer object which was allocated through *cma_alloc*.

- **src** (`cffi.FFI.CData`) – Source buffer object which was allocated through *cma_alloc*.

- **nbytes** (`int`) – Number of bytes to copy.

> **Returns**

> **Return type** None

**cma_stats**()
    Get current CMA memory Stats.

    *CMA Memory Available* : Systemwide CMA memory availability.

    *CMA Memory Usage* : CMA memory used by current object.

    *Buffer Count* : Buffers allocated by current object.

> **Returns** Dictionary of current stats.

> **Return type** dict

**ffi = <cffi.api.FFI object>**

**flush**(*bo*, *offset*, *vaddr*, *nbytes*)

**invalidate**(*bo*, *offset*, *vaddr*, *nbytes*)

**libxlnk = None**

**libxlnk_path = '/usr/lib/libcma.so'**

---

> **classmethod set_allocator_library**(*path*)
>> Change the allocator used by Xlnk instances
>>
>> This should only be called when there are no allocated buffers - using or freeing any pre-allocated buffers after calling this function will result in undefined behaviour. This function is needed for SDx based designs where it is desired that PYNQ and SDx runtime share an allocator. In this case, this function should be called with the SDx compiled shared library prior to any buffer allocation
>>
>> If loading of the library fails an exception will be raised, Xlnk.libxlnk_path will be unchanged and the old allocator will still be in use.
>>
>>> Parameters **path** (*str*) – Path to the library to load

> **xlnk_reset**()
>> Systemwide Xlnk Reset.

### Notes

> This method resets all the CMA buffers allocated across the system.
>
>> **Returns**
>>
>> **Return type** None

pynq.xlnk.**sig_handler**(*signum*, *frame*)

## 2.9 Verification

This section documents the test infrastructure supplied with the *pynq* package. It is organized as follows:

- *Running Tests* : describes how to run the pytest.
- *Writing Tests* : explains how to write tests.
- *Miscellaneous* : covers additional information relating to tests.

### 2.9.1 Running Tests

The *pynq* package provides tests for most python modules.

To run all the tests together, pytest can be run in a Linux terminal on the board. All the tests will be automatically collected in the current directory and child directories.

---

**Note:** The pytests have to be run as root

---

To run all the collected tests in a single shot:

```
cd /home/xilinx/pynq
sudo py.test -vsrw
```

For any given board, it is possible not to be able to use all the software drivers. For such cases, it is more common to run tests in a specific folder:

```
cd /home/xilinx/pynq/<driver_folder>
sudo py.test -vsrw
```

For a complete list of pytest options, please refer to Usage and Invocations - Pytest.

### Collection Phase

During this phase, the pytest will collect all the test modules in the current directory and all of its child directories. The user will be asked to confirm the tests.

For example:

```
Test trace analyzers? ([yes]/no)>>> yes
```

For the answer to such a question, "yes", "YES", "Yes", "y", and "Y" are acceptable; the same applies for "no" as an answer. You can also press *Enter*; this is equivalent to "yes".

Answering "No" will skip the corresponding test(s) during the testing phase.

Sometimes a device connected to the board will be required before the test.

```
Pmod OLED attached to the board? ([yes]/no)>>> yes
Type in the interface ID of the Pmod OLED (PMODA/PMODB):
```

For such a question, users need to type in the options specified inside the parentheses.

### Testing Phase

The test suite will guide the user through all the tests implemented in the pynq package. As part of the tests, the user will be prompted for confirmation that the tests have passed, for example:

```
test_leds_on_off ...
Onboard LED 0 on? ([yes]/no)>>>
```

Again press "Enter", or type "yes", "no" etc.

At the end of the testing phase, a summary will be given to show users how many tests are passed / skipped / failed.

## 2.9.2 Writing Tests

This section follows the guide available on Pytest Usages and Examples. You can write a test class with assertions on inputs and outputs to allow automatic testing. The names of the test modules *must* start with *test_*; all the methods for tests in any test module *must* also begin with *test_*. One reason to enforce this is to ensure the tests will be collected properly. See the Full pytest documentation for more details.

### Step 1

First of all, the pytest package has to be imported:

```python
import pytest
```

### Step 2

Decorators can be specified directly above the methods. For example, users can specify (1) the order of this test in the entire pytest process, and (2) the condition to skip the corresponding test. More information on decorators can be found in Marking test functions with attributes - Pytest.

An example will be given in the next step.

### Step 3

Directly below decorators, you can write some assertions/tests. See the example below:

```python
@pytest.mark.run(order=1)
def test_superuser():
"""Test whether the user have the root privilege.

Note
----
To pass all of the pytests, need the root access.

"""
assert os.geteuid() == 0, "Need ROOT access in order to run tests."
```
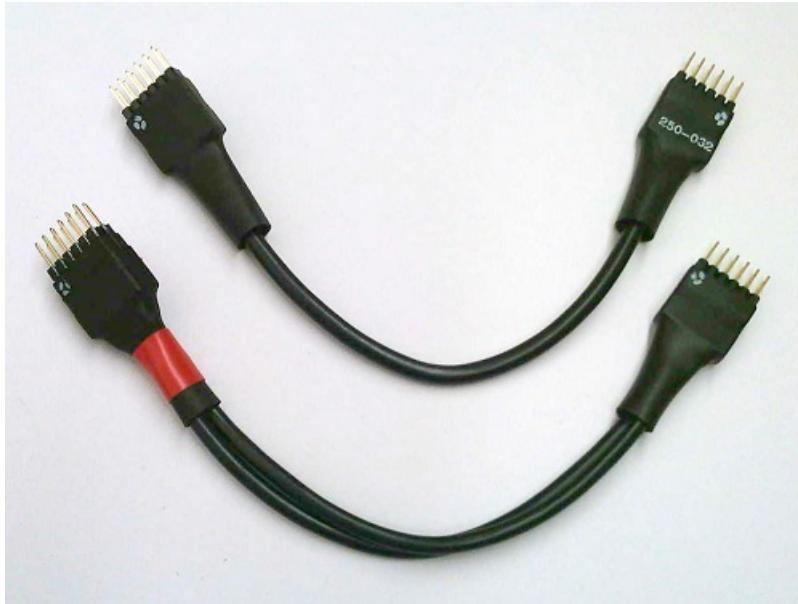
Note the *assert* statements specify the desired condition, and raise exceptions whenever that condition is not met. A customized exception message can be attached at the end of the *assert* methods, as shown in the example above.

## 2.9.3 Miscellaneous Test Setup

Some tests may require users to leverage jumper wires and external breadboard. Our pytest suite will provide some instructions for users to follow.

In some cases, two types of cables are used with the tests:



- *Straight cable* (upper one in the image): The internal wires between the two ends are straight. This cable is intended for use as an extension cable.

- *Loopback cable* (lower one in the image, with red ribbon): The internal wires are twisted. This cable is intended for testing.

There are marks on the connectors at each end of the cable to indicate the orientation and wiring of the cable.

**Note:** You must not short VCC and GND as it may damage the board. It is good practice to align the pins with the dot marks to VCC of the Pmod interfaces.

**Note:** For testing, there is only one connection type (mapping) allowed for each cable type. Otherwise VCC and GND could be shorted, damaging the board.

# 2.10 Frequently Asked Questions (FAQs)

## 2.10.1 Troubleshooting

### I can't connect to my board

1. Check the board is powered on and that the bitstream has been loaded (Most boards have a "DONE" LED to indicate this)

2. Your board and PC/laptop must be on the same network, or have a direct network connection. Check that you can *ping* the board (hostname, or IP address) from a command prompt or terminal on your host PC:

```
ping pynq
```

or

```
ping 192.168.2.99
```

(The default IP address of the board is : 192.168.2.99)

3. Log on to the board through a terminal, and check the system is running, i.e. that the Linux shell is accessible. See below for details on logging on with a terminal.

4. If you can't ping the board, or the host PC, check your network settings.

   - You must ensure your PC/laptop and board have IP addresses in the same range. If your network cables are connected directly to your PC/laptop and board, you may need to set a static IP address for your PC/laptop manually. See *Assign your computer a static IP address*.

   - If you have a proxy setup, you may need to add a rule to bypass the board hostname/ip address.

   - If you are using a docking station, when your laptop is docked, the Ethernet port on the PC may be disabled.

### My Pynq-Z1/Z2 board is not powering on (No Red LED)

The board can be powered by USB cable, or power adapter (7 - 15V V 2.1mm centre-positive barrel jack). Make sure Jumper JP5 is set to USB or REG (for power adapter). If powering the board via USB, make sure the USB port is fully powered. Laptops in low power mode may reduce the available power to a USB port.

### The bitstream is not loading (No Green LED)

   - Check the Micro-SD card is inserted correctly (the socket is spring loaded, so push it in until you feel it click into place).

   - Check jumper JP4 is set to SD (board boots from Micro SD card).

- Connect a terminal and verify that the Linux boot starts.

If the Linux boot does not start, or fails, you may need to (re)flash the PYNQ image to the Micro SD card.

### The hostname of the board is not resolving/not found

It may take the hostname (pynq) some time to resolve on your network. If you know the IP address of the board, it may be faster to use the IP address to navigate to the Jupyter portal instead of the hostname.

For example, in your browser, go to http://192.168.2.99:9090 if the board is using the static IP address 192.168.2.99.

You can find the IP by first connecting a terminal to the board, then running the following command in Linux command line:

```
ifconfig
```

Check the settings for *eth0* and look for an IP address.

### I don't have an Ethernet port on my PC/Laptop

If you don't have an Ethernet port, you can get a USB to Ethernet adapter.

If you have a wireless router with Ethernet ports (LAN), you can connect your board to an Ethernet port on your router, and connect to it from your PC using WiFi. (You may need to change settings on your Router to enable the Wireless network to communicate with your LAN - check your equipment documentation for details.)

You can also connect a WiFi dongle to the board, and set up the board to connect to the wireless network. Your host PC can then connect to the same wireless network to connect to the board.

### How can I enable wireless access point?

If the board has a supported built-in wireless module, and the PYNQ image has the *wpa_ap* package installed, you can try start the WiFi access point as follows. Note that *wpa_ap.service* is disabled by default.

To check if the WiFi access point service is available:

```
systemctl list-unit-files | grep wpa_ap.service
```

To start the service immediately:

```
sudo systemctl start wpa_ap.service
```

To enable the service for each boot:

```
sudo systemctl enable wpa_ap.service
```

Similarly, you can use *stop* or *disable* to revert the above commands.

### How do I setup my computer to connect to the board?

If you are connecting your board to your network (i.e. you have plugged the Ethernet cable into the board, and the other end into a network switch, or home router), then you should not need to setup anything on your computer. Usually, both your computer, and board will be assigned an IP address automatically, and they will be able to communicate with each other.

If you connect your board directly to your computer with an ethernet cable, then you need to make sure that they have IP addresses in the same range. The board will assign itself a static IP address (by default 192.168.2.99), and you will need to assign a static IP address in the same range to the computer. This allows your computer and board to communicate to each other over the Ethernet cable.

See *Assign your computer a static IP address*.

### I can't connect to the Jupyter portal!

If your board is powered on, and you see the Red and Green LEDs, but still can't connect to the Jupyter Portal, or see the Samba shared drive, then you need to verify your IP adddresses.

By default, the board has DHCP enabled. If you plug the board into a home router, or network switch connected to your network, it should be allocated an IP address automatically. If not, it should fall back to a static IP address of *192.168.2.99*.

If you plug the Ethernet cable directly to your computer, you will need to configure your network card to have an IP in the same address range, e.g. *192.168.2.1*.

### VPN

If your PC/laptop is connected to a VPN, and your board is not on the same VPN network, this will block access to local IP addresses. You need to disable the VPN, or set it to bypass the board address.

### Proxy

If your board is connected to a network that uses a proxy, you need to set the proxy variables on the board

```
set http_proxy=my_http_proxy:8080
set https_proxy=my_https_proxy:8080
```

## 2.10.2 Board/Jupyter settings

### How do I modify the board settings?

Linux is installed on the board. Connect to the board using a terminal, and change the settings as you would for any other Linux machine.

### How do I find the IP address of the board?

Connect to the board using a terminal (see above) and type:

```
hostname -I
```

This will help you find the IP address for the eth0 Ethernet adapter or the WiFi dongle.

### How do I set/change the static IP address on the board?

The Static IP address is set in `/etc/dhcp/dhclient.conf` - you can modify the board's static IP here.

---

### How do I find my hostname?

Connect to the board using a terminal and run:

```
hostname
```

### How do I change the hostname?

If you have multiple boards on the same network, you should give them different host names. You can change the hostname by editing the Linux hostname files: `/etc/hostname` and `/etc/hosts`.

### What is the user account and password?

The username for all Linux, Jupyter and Samba logins is `xilinx`. The password is `xilinx`. For vagrant Ubuntu VM, both the username and password are `vagrant`.

### How do I enable/disable the Jupyter notebook password?

The Jupyter configuration file can be found at

```
/root/.jupyter/jupyter_notebook_config.py
```

You can add or comment out the c.NotebookApp.password to bypass the password authentication when connecting to the Jupyter Portal.

```
c.NotebookApp.password =u'sha1:6c2164fc2b22:
↪ed55ecf07fc0f985ab46561483c0e888e8964ae6'
```

### How do I change the Jupyter notebook password

A hashed password is saved in the Jupyter Notebook configuration file.

```
/root/.jupyter/jupyter_notebook_config.py
```

You can create a hashed password using the function `IPython.lib.passwd()`:

```python
from IPython.lib import passwd
password = passwd("secret")
6c2164fc2b22:ed55ecf07fc0f985ab46561483c0e888e8964ae6
```

You can then add or modify the line in the *jupyter_notebook_config.py* file

```
c.NotebookApp.password =u'sha1:6c2164fc2b22:
↪ed55ecf07fc0f985ab46561483c0e888e8964ae6'
```

## 2.10.3 General Questions

### Does PYNQ support Python 2.7?

The PYNQ image is based on Ubuntu which includes Python 2.7 in the root file system. The Python package *pynq*, however, is based on Python 3.6; this python package is not compatible with Python 2.7.

### Where can I find the overlay bitstreams?

In order to keep a reasonable Github repository size, starting from image v2.5, we no longer store bitstreams in our Github repository. Instead, we provide a simple script allowing users to build the bitstreams by themselves. This script (*build.sh*) is located at the root of the PYNQ repository. To run this script, make sure you have Vivado and SDK installed on your Ubuntu machine, and run:

```
./build.sh
```

If you are using our SD build flow, this step will be run automatically.

### Where can I find the MicroBlaze bin files?

In order to keep a reasonable Github repository size, starting from image v2.5, we no longer store compiled MicroBlaze binaries in our Github repository. Instead, we provide a simple script allowing users to build the binaries by themselves. This script (*build.sh*) is located at the root of the PYNQ repository. To run this script, make sure you have Vivado and SDK installed on your Ubuntu machine, and run:

```
./build.sh
```

If you are using our SD build flow, this step will be run automatically.

### How do I write the Micro SD card image?

You can find instructions in *Writing the SD Card Image*.

### What type of Micro SD card do I need?

We recommend you use a card at least 8GB in size and at least class 4 speed rating.

### How do I connect to the board using a terminal on Windows?

To do this, you need to connect to the board using a terminal:

Connect a Micro USB cable to the board and your computer, and use a terminal emulator (puTTY, TeraTerm etc) to connect to the board.

> Terminal Settings:
>
> - 115200 baud
> - 8 data bits
> - 1 stop bit
> - No Parity
> - No Flow Control

### How do I connect to the board using a terminal on Mac OS/Linux?

Open a Terminal window on MacOS or an XTerm (or your favorite terminal program) on Linux.

Issue the following command to view current serial devices.

```
ls /dev/cu.usb*
```

Connect a Micro USB cable to the board and your computer.

Issue the following command again to identify the device.

```
ls /dev/cu.usb*
```

Identify the change of items in the list, and issue the following command:

```
screen /dev/<device> 115200 -L
```

For example, if the difference was *cu.usbmodem0004*, the command would be:

```
screen /dev/cu.usbmodem0004 115200 -L
```

## 2.11 Glossary

### 2.11.1 A-G

**APSOC** All Programmable System on Chip

**BSP** A board support package (BSP) is a collection of low-level libraries and drivers. The Xilinx® software development Kit (SDK) uses a BSP to form the lowest layer of your application software stack. Software applications must link against or run on top of a given software platform using the APIs that it provides. Therefore, before you can create and use software applications in SDK, you must create a board support package

**FPGA** Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks.

### 2.11.2 H-R

**HDF** Hardware Definition File (.hdf). This file is created by Vivado and contains information about a processor system in an FPGA overlay. The HDF specifies the peripherals that exist in the system, and the memory map. This is used by the BSP to build software libraries to support the available peripherals.

**I2C** See IIC

**IIC** Inter-Integrated Circuit; multi-master, multi-slave, single-ended, serial computer bus protocol

**IOP** Input/Output Processor.

**Jupyter (Notebooks)** Jupyter is an open source project consisting of an interactive, web application that allows users to create and share notebook documents that contain live code and the full range of rich media supported by modern browsers. These include text, images, videos, LaTeX-styled equations, and interactive widgets. The Jupyter framework is used as a front-end to over 40 different programming languages. It originated from the interactive data science and scientific computing communities. Its uses include: data cleaning and transformation, numerical simulation, statistical modelling, machine learning and much more.

**MicroBlaze** MicroBlaze is a soft microprocessor core designed for Xilinx FPGAs. As a soft-core processor, MicroBlaze is implemented entirely in the general-purpose memory and logic fabric of an FPGA.

**Pmod Interface** The Pmod or Peripheral Module interface is used to connect low frequency, low I/O pin count peripheral modules to host controller boards.accessory boards to add functionality to the platform. e.g. ADC, DAC, I/O interfaces, sensors etc.

**(Micro) SD** Secure Digital (Memory Card standard)

**readthedocs.org** readthedocs.org is a popular website that hosts the documentation for open source projects at no cost. readthedocs.org uses Sphinx document generation tools to automatically generate both the website and PDF versions of project documentation from a GitHub repository when new updates are pushed to that site.

**REPL** A read–eval–print loop (REPL), also known as an interactive toplevel or language shell, is a simple, interactive computer programming environment that takes single user inputs (i.e. single expressions), evaluates them, and returns the result to the user; a program written in a REPL environment is executed piecewise. The term is most usually used to refer to programming interfaces similar to the classic Lisp machine interactive environment. Common examples include command line shells and similar environments for programming languages, and is particularly characteristic of scripting languages wikipedia

**reST** Restructured text is a markup language used extensively with the Sphinx document generator

### 2.11.3 S-Z

**SDK** Xilinx SDK - Software Development Kit. Software development environment including cross-compiles for ARM®, and MicroBlaze processors. Also includes debug, and profiling tools. Required to build software for a MicroBlaze processor inside an IOP.

**SOC** System On Chip

**Sphinx** A document generator written in Python and used extensively to document Python and other coding projects

**SPI** Serial Peripheral Interface; synchronous serial communication interface specification

**UART** Universal asynchronous receiver/transmitter; Serial communication protocol

**Vivado** Vivado Design Suite is a suite of computer-aided design tools provided by Xilinx for creating FPGA designs. It is used to design and implement the overlays used in Pynq.

**XADC** An XADC is a hard IP block that consists of dual 12-bit, 1 Mega sample per second (MSPS), analog-to-digital converters and on-chip sensors which are integrated into Xilinx 7 series FPGA devices

**Zynq®** Zynq-7000 All Programmable SoC (APSoC) devices integrate the software programmability of an ARM®-based processor with the hardware programmability of an FPGA, enabling key analytics and hardware acceleration while integrating CPU, DSP, ASSP, and mixed signal functionality on a single device. Zynq-7000 AP SoCs infuse customizable intelligence into today's embedded systems to suit your unique application requirements

**Zynq PL** Programmable Logic - FPGA fabric

**Zynq PS** Processing System - SOC processing subsystem built around dual-core, ARM Cortex-A9 processor

## 2.12 Useful Links

### 2.12.1 Git

- Interactive introduction to Git
- Free PDF copy of The Pro Git book by Scott Chacon and Ben Straub

### 2.12.2 Jupyter

- Jupyter Project
- Try Jupyter in your browser

### 2.12.3 PUTTY (terminal emulation software)

- PUTTY download page

### 2.12.4 Pynq Technical support

- Pynq Technical support

### 2.12.5 Python built-in functions

- C Python native functions

### 2.12.6 Python training

- The Python Tutorial from the Python development team
- Google Python training including videos
- Python Tutor including visualization of Python program execution
- 20 Best Free Tutorials to Learn Python as of 9 Oct 2015

### 2.12.7 reStructuredText

- reStructuredText docs
- reStructuredText Primer
- Online reStructuredText editor

### 2.12.8 Sphinx

- The official Sphinx docs
- Online reST and Sphinx editor with rendering
- A useful Sphinx cheat sheet

- Jupyter Notebook Tools for Sphinx

## 2.13 Appendix

### 2.13.1 Technology Backgrounder

#### Overlays and Design Re-use

The 'magic' of mapping an application to an APSoC, without designing custom hardware, is achieved by using *FPGA overlays*. FPGA overlays are FPGA designs that are both highly configurable and highly optimized for a given domain. The availability of a suitable overlay removes the need for a software designer to develop a new bitstream. Software and system designers can customize the functionality of an existing overlay *in software* once the API for the overlay bitstream is available.

An FPGA overlay is a domain-specific FPGA design that has been created to be highly configurable so that it can be used in as many different applications as possible. It has been crafted to maximize post-bitstream programmability which is exposed via its API. The API provides a new entry-point for application-focused software and systems engineers to exploit APSoCs in their solutions. With an API they only have to write software to program configure the functions of an overlay for their applications.

By analogy with the Linux kernel and device drivers, FPGA overlays are designed by relatively few engineers so that they can be re-used by many others. In this way, a relatively small number of overlay designers can support a much larger community of APSoC designers. Overlays exist to promote re-use. Like kernels and device drivers, these hardware-level artefacts are not static, but evolve and improve over time.

#### Characteristics of Good Overlays

Creating one FPGA design and its corresponding API to serve the needs of many applications in a given domain is what defines a successful overlay. This, one-to-many relationship between the overlay and its users, is different from the more common one-to-one mapping between a bitstream and its application.

Consider the example of an overlay created for controlling drones. Instead of creating a design that is optimized for controlling just a single type of drone, the hardware architects recognize the many common requirements shared by different drone controllers. They create a design for controlling drones that is a flexible enough to be used with several different drones. In effect, they create a drone-control overlay. They expose, to the users of their bitstream, an API through which the users can determine in software the parameters critical to their application. For example, a drone control overlay might support up to eight, pulse-width-modulated (PWM) motor control circuits. The software programmer can determine how many of the control circuits to enable and how to tune the individual motor control parameters to the needs of his particular drone.

The design of a good overlay shares many common elements with the design of a class in object-oriented software. Determining the fundamental data structure, the private methods and the public interface are common requirements. The quality of the class is determined both by its overall usefulness and the coherency of the API it exposes. Well-engineered classes and overlays are inherently useful and are easy to learn and deploy.

Pynq adopts a holistic approach by considering equally the design of the overlays, the APIs exported by the overlays, and how well these APIs interact with new and existing Python design patterns and idioms to simplify and improve the APSoC design process. One of the key challenges is to identify and refine good abstractions. The goal is to find abstractions that improve design coherency by exposing commonality, even among loosely-related tasks. As new overlays and APIs are published, we expect that the open-source software community will further improve and extend them in new and unexpected ways.

Note that FPGA overlays are not a novel concept. They have been studied for over a decade and many academic papers have been published on the topic.

**The Case for Productivity-layer Languages**

Successive generations of All Programmable Systems on Chip embed more processors and greater processing power. As larger applications are integrated into APSoCs, the embedded code increases also. Embedded code that is speed or size critical, will continue to be written in C/C++. These 'efficiency-layer or systems languages' are needed to write fast, low-level drivers, for example. However, the proportion of embedded code that is neither speed-critical or size-critical, is increasing more rapidly. We refer to this broad class of code as *embedded applications code*.

Programming embedded applications code in higher-level, 'productivity-layer languages' makes good sense. It simply extends the generally-accepted best-practice of always programming at the highest possible level of abstraction. Python is currently a premier productivity-layer language. It is now available in different variants for a range of embedded systems, hence its adoption in Pynq. Pynq runs CPython on Linux on the ARM® processors in Zynq® devices. To further increase productivity and portability, Pynq uses the Jupyter Notebook, an open-source web framework to rapidly develop systems, document their behavior and disseminate the results.

## 2.13.2 Writing the SD Card Image

### Windows

- Insert the Micro SD card into your SD card reader and check which drive letter was assigned. You can find this by opening Computer/My Computer in Windows Explorer.

- Download the Win32DiskImager utility from the Sourceforge Project page

- Extract the *Win32DiskImager* executable from the zip file and run the Win32DiskImager utility as administrator. (Right-click on the file, and select Run as administrator.)

- Select the PYNQ-Z1 image file (.img).

- Select the drive letter of the SD card. Be careful to select the correct drive. If you select the wrong drive you can overwrite data on that drive. This could be another USB stick, or memory card connected to your computer, or your computer's hard disk.

- Click **Write** and wait for the write to complete.

### MAC / OS X

On Mac OS, you can use dd, or the graphical tool ImageWriter to write to your Micro SD card.

- First open a terminal and unzip the image:

```
unzip pynq_z1_image_2016_09_14.zip -d ./
```

### ImageWriter

Note the Micro SD card must be formatted as FAT32.

- Insert the Micro SD card into your SD card reader

- From the Apple menu, choose "About This Mac", then click on "More info…"; if you are using Mac OS X 10.8.x Mountain Lion or newer, then click on "System Report".

- Click on "USB" (or "Card Reader" if using a built-in SD card reader) then search for your SD card in the upper-right section of the window. Click on it, then search for the BSD name in the lower-right section; it will look something like **diskn** where n is a number (for example, disk4). Make sure you take a note of this number.

- Unmount the partition so that you will be allowed to overwrite the disk. To do this, open Disk Utility and unmount it; do not eject it, or you will have to reconnect it. Note that on Mac OS X 10.8.x Mountain Lion, "Verify Disk" (before unmounting) will display the BSD name as */dev/disk1s1* or similar, allowing you to skip the previous two steps.

- From the terminal, run the following command:

```
sudo dd bs=1m if=path_of_your_image.img of=/dev/rdiskn
```

Remember to replace n with the number that you noted before!

If this command fails, try using disk instead of rdisk:

```
sudo dd bs=1m if=path_of_your_image.img of=/dev/diskn
```

Wait for the card to be written. This may take some time.

### Command Line

- Open a terminal, then run:

```
diskutil list
```

- Identify the disk (not partition) of your SD card e.g. disk4, not disk4s1.

- Unmount your SD card by using the disk identifier, to prepare for copying data to it:

```
diskutil unmountDisk /dev/disk<disk# from diskutil>
```

  where disk is your BSD name e.g. *diskutil unmountDisk /dev/disk4*

- Copy the data to your SD card:

```
sudo dd bs=1m if=image.img of=/dev/rdisk<disk# from diskutil>
```

  where disk is your BSD name e.g. sudo dd bs=1m if=pynq_z1_image_2016_09_07.img of=/dev/rdisk4

This may result in a dd: invalid number '1m' error if you have GNU coreutils installed. In that case, you need to use a block size of 1M in the bs= section, as follows:

```
sudo dd bs=1M if=image.img of=/dev/rdisk<disk# from diskutil>
```

Wait for the card to be written. This may take some time. You can check the progress by sending a SIGINFO signal (press Ctrl+T).

If this command still fails, try using disk instead of rdisk, for example:

```
sudo dd bs=1m if=pynq_z1_image_2016_09_07.img of=/dev/disk4
```

### Linux

### dd

Please note the dd tool can overwrite any partition on your machine. Please be careful when specifying the drive in the instructions below. If you select the wrong drive, you could lose data from, or delete your primary Linux partition.

- Run *df -h* to see what devices are currently mounted.

- Insert the Micro SD card into your SD card reader

- Run df -h again.

The new device that has appeared is your Micro SD card. The left column gives the device name; it will be listed as something like /dev/mmcblk0p1 or /dev/sdd1. The last part (p1 or 1 respectively) is the partition number but you want to write to the whole SD card, not just one partition. You need to remove that part from the name. e.g. Use /dev/mmcblk0 or /dev/sdd as the device name for the whole SD card.

Now that you've noted what the device name is, you need to unmount it so that files can't be read or written to the SD card while you are copying over the SD image.

- Run *umount /dev/sdd1*, replacing sdd1 with whatever your SD card's device name is (including the partition number).

If your SD card shows up more than once in the output of df due to having multiple partitions on the SD card, you should unmount all of these partitions.

- In the terminal, write the image to the card with the command below, making sure you replace the input file if= argument with the path to your .img file, and the /dev/sdd in the output file of= argument with the right device name. This is very important, as you will lose all data on the hard drive if you provide the wrong device name. Make sure the device name is the name of the whole Micro SD card as described above, not just a partition of it; for example, sdd, not sdds1, and mmcblk0, not mmcblk0p1.

*sudo dd bs=4M if=pynq_z1_image_2016_09_07.img of=/dev/sdd*

Please note that block size set to 4M will work most of the time; if not, please try 1M, although this will take considerably longer.

The dd command does not give any information of its progress and so may appear to have frozen; it could take a few minutes to finish writing to the card.

Instead of dd you can use *dcfldd*; it will give a progress report about how much has been written.

### 2.13.3 Assign your computer a static IP address

Instructions may vary slightly depending on the version of operating system you have. You can also search on google for instructions on how to change your network settings.

You need to set the IP address of your laptop/pc to be in the same range as the board. e.g. if the board is 192.168.2.99, the laptop/PC can be 192.168.2.x where x is 0-255 (excluding 99, as this is already taken by the board).

You should record your original settings, in case you need to revert to them when finished using PYNQ.

**Windows**

- Go to Control Panel -> Network and Internet -> Network Connections

- Find your Ethernet network interface, usually *Local Area Connection*

- Double click on the network interface to open it, and click on *Properties*

- Select Internet Protocol Version 4 (TCP/IPv4) and click *Properties*

- Select *Use the following IP address*

- Set the Ip address to 192.168.2.1 (or any other address in the same range as the board)

- Set the subnet mask to 255.255.255.0 and click **OK**

**Mac OS**

- Open *System Preferences* then open *Network*
- Click on the connection you want to set manually, usually *Ethernet*
- From the Configure IPv4 drop down choose Manually
- Set the IP address to 192.168.2.1 (or any other address in the same range as the board)
- Set the subnet mask to 255.255.255.0 and click **OK**

The other settings can be left blank.

**Linux**

- Edit this file (replace gedit with your preferred text editor):

      sudo gedit /etc/network/interfaces

The file usually looks like this:

```
auto lo eth0
iface lo inet loopback
iface eth0 inet dynamic
```

- Make the following change to set the eth0 interface to the static IP address 192.168.2.1

```
iface eth0 inet static
    address 192.168.2.1
    netmask 255.255.255.0
```

Your file should look like this:

```
auto lo eth0
iface lo inet loopback
iface eth0 inet static
        address 192.168.2.1
        netmask 255.255.255.0
```

## 2.14 Change Log

### 2.14.1 Version 2.5

- **Image releases:**
    - pynq_z1_v2.5
    - pynq_z2_v2.5
    - zcu104_v2.5
    - zcu111_v2.5

Within those image files, PYNQ v2.5 is already installed. Updates to PYNQ since the last release include:

- **Productivity Additions**
    - Updated to JupyterLab 1.1.3

– JupyterLab extensions support added

– Support for multiple memories using mem_dict entries

– Support for Device Tree Overlays delivered with PL overlays

– Support for custom PL device communication using the Device metaclass

- **Programmable Logic Updates**

  – All bitstreams built using Vivado 2019.1

  – XRT Support added (beta)

- **Repository Updates**

  – Jenkins CI added

  – Sdist support added (removing all binaries from the repository)

- **SDBuild Updates**

  – Boot partition built on Petalinux 2019.1

### 2.14.2 Version 2.4

- **Image releases:**

  – pynq_z1_v2.4

  – pynq_z2_v2.4

  – zcu104_v2.4

  – zcu111_v2.4

Documentation updated 22 Feb 2019

- **Board Additions**

  – RFSoC support added in the new ZCU111-PYNQ repository

- **Programmable Logic Updates**

  – All bitstreams built using Vivado 2018.3

  – Partial reconfiguration support added (beta)

  – Expanded metadata parsing using the Vivado hwh files

- **SDBuild Updates**

  – Boot partition built on Petalinux 2018.3

  – SDSoC 2018.3 support added

  – Vagrant configuration file for users building their own SDCard images

  – Yocto recipes added for including PYNQ in Petalinux root filesystems

### 2.14.3 Version 2.3

- **Image releases:**

  – pynq_z1_v2.3

  – pynq_z2_v2.3

&ndash; zcu104_v2.3

Documentation updated 7 Sep 2018

- **Architecture Additions**

    &ndash; Zynq UltraScale+ (ZU+) support added

- **Board Additions**

    &ndash; ZCU104 support added

- **Programmable Logic Updates**

    &ndash; All bitstreams built using Vivado 2018.2

    &ndash; Initial support for DSA generation and PL parsing added

    &ndash; Removed custom toplevel wrapper file requirement

- **SDBuild Updates**

    &ndash; Root filesystem based on Ubuntu 18.04

    &ndash; Boot partition built on Petalinux 2018.2

    &ndash; SDSoC 2018.2 support added

    &ndash; Added fpga_manager support for Zynq and ZU+

    &ndash; AWS Greengrass kernel configuration options added

    &ndash; Custom board support updated

- **New PYNQ Python Modules**

    &ndash; Added ZU+ DisplayPort

    &ndash; Added PMBus power monitoring

    &ndash; Added uio support

    &ndash; Added AXI IIC support

- **New Microblaze Programming Notebooks**

    &ndash; Added arduino ardumoto, arduino joystick, grove usranger notebooks

### 2.14.4 Version 2.2

Image release: pynq_z2_v2.2

Documentation updated 10 May 2018

- **Board Additions**

    &ndash; PYNQ-Z2 support added

- **New Microblaze Subsystems**

    &ndash; Added RPi Microblaze subsystem, bsp and notebooks

- **New IP**

    &ndash; Added audio with codec support

## 2.14.5 Version 2.1

Image release: pynq_z1_v2.1

Documentation updated 21 Feb 2018

- **Overlay Changes**

    - All overlays updated to build with Vivado 2017.4

    - Hierarchical IPs' port names refactored for readability and portability

    - The IOP hierarchical blocks are renamed from iop_1, 2, 3 to iop_pmoda, iop_pmodb, and iop_arduino

    - The Microblaze subsystem I/O controllers were renamed to be iop agnostic

- **Base Overlay Changes**

    - The onboard switches and LEDs controlled are now controlled by two AXI_GPIO IPs.

    - The 2nd I2C (shared) from the Arduino IOP was removed

- **IP Changes**

    - IP refactored for better portability to new boards and interfaces

    - IO Switch now with configuration options for pmod, arduino, dual pmod, and custom I/O connectivity

    - IO Switch now with standard I/O controller interfaces for IIC and SPI

- **Linux changes**

    - Updated to Ubuntu 16.04 LTS (Xenial)

    - Updated kernel to tagged 2017.4 Xilinx release.

    - Jupyter now listens on both :80 and :9090 ports

    - opencv2.4.9 removed

- **Microblaze Programming**

    - IPython magics added for Jupyter programming of Microblazes

    - Microblaze pyprintf, RPC, and Python-callable function generation added.

    - New notebooks added to demonstrate the programming APIs

- **Repository Changes**

    - Repository pynqmicroblaze now a package to support Microblaze programming

- **Pynq API Changes**

    - Audio class renamed to AudioDirect to allow for future audio codec classes

- **New Python Packages**

    - netifaces, imutils, scikit-image

- **Device Updates**

    - Removed version-deprecated Grove-I2C Color Sensor

## 2.14.6 Version 2.0

Image release: pynq_z1_v2.0

Documentation updated: 18 Aug 2017

- **Overlay changes**

  - New logictools overlay

  - Updated to new Trace Analyzer IP in the base overlay

- **Repository Changes**

  - Repository restructured to provide better support for multiple platforms

  - **Repository now supports direct pip install**

    * update_pynq.sh is now deprecated

- PYNQ Image build flow now available

- **Pynq API Changes**

  - pynq.lib combines previous packages: pynq.board, pynq.iop, pynq.drivers

  - The pynq.iop subpackage has been restructured into lib.arduino and lib.pmod

    For example:

    ```
    from pynq.iop import Arduino_Analog
    ```

    is replaced by:

    ```
    from pynq.lib.arduino import Arduino_Analog
    ```

  - Overlay() automatically downloads an overlays on instantiation by default. Explicit .download() is not required

  - DMA driver replaced with new version

    The buffer is no longer owned by the DMA driver and should instead be allocated using *Xlnk.cma_array*. Driver exposes both directions of the DMA engine. For example:

    ```
    send_buffer = xlnk.cma_array(1024, np.float32)
    dma.sendchannel.transfer(send_buffer)
    dma.wait()
    # wait dma.wait_async() also available in coroutines
    ```

  - New Video subsystem with support for openCV style frame passing, color space transforms, and grayscale conversion

  - New PynqMicroblaze parent class to implement any PYNQ MicroBlaze subsystem

  - New DefaultIP driver to access MMIO, interrupts and GPIO for an IP and is used as the base class for all IP drivers

  - New DefaultHierarchy driver to access contained IP as attributes and is used as the base class for all hierarchy drivers

  - New AxiGPIO driver

- **Linux changes**

  - Addition USB Ethernet drivers added

- – Start-up process added to systemd services

- **New Python Packages**

  - – cython

- **IP changes**

  - – Updated Trace Analyzer, deprecated Trace Buffer

  - – Updated Video subsytem with added HLS IP to do color space transforms, and grayscale conversion

  - – Added new logictools overlay IP: Pattern Generator, Boolean Generator, FSM Generator

- **Documentation changes**

  - – Restructured documentation

  - – Added *PYNQ Overlays* section describing each overlay and its hardware components

  - – Added *PYNQ Libraries* section descriping Python API for each hardware component

  - – Added *pynq Package* section for Python Docstrings

  - – Creating Overlays section renamed to *Overlay Design Methodology*

  - – Added *PYNQ SD Card* section describing PYNQ image build process

### 2.14.7 Version 1.4

Image release: pynq_z1_image_2016_02_10

Documentation updated: 10 Feb 2017

- Xilinx Linux kernel upgraded to 4.6.0

- **Added Linux Packages**

  - – Python3.6

  - – iwconfig

  - – iwlist

  - – microblaze-gcc

- **New Python Packages**

  - – asyncio

  - – uvloop

  - – transitions

  - – pygraphviz

  - – pyeda

- **Updated Python Packages**

  - – pynq

  - – Jupyter Notebook Extension added

  - – IPython upgraded to support Python 3.6

  - – pip

- **Other changes**

- Jupyter extensions

- reveal.js updated

- update_pynq.sh

- wavedrom.js

- **Base overlay changes**

    - IOP interface to DDR added (Pmod and Arduino IOP)

    - Interrupt controller from overlay to PS added. IOP GPIO connected to interrupt controller.

    - Arduino GPIO base address has changed due to merge of GPIO into a single block. *arduino_grove_ledbar* and *arduino_grove_buzzer* compiled binaries are not backward compatible with previous Pynq overlay/image.

- **Pynq API/driver changes**

    - TraceBuffer: Bit masks are not required. Only pins should be specified.

    - PL: `pl_dict` returns an integer type for any base address http://pynq.readthedocs.io/en/latest/4_programming_python.html / address range.

    - Video: Video mode constants are exposed outside the class.

    - Microblaze binaries for IOP updated.

    - Xlnk() driver updated, with better support for SDX 2016.3. Removed the customized Xlnk() drivers and use the libsds version.

- **Added new iop modules**

    - arduino_lcd18

- **Added Notebooks**

    - audio (updated)

    - arduino_lcd (new)

    - utilities (new)

    - asyncio (new)

- **Documentation changes**

    - New section on peripherals and interfaces

    - New section on using peripherals in your applications

    - New section on Asyncio/Interrupts

    - New section on trace buffer

### 2.14.8 Version 1.3

Image release: pynq_z1_image_2016_09_14

Documentation updated: 16 Dec 2016

- **Added new iop modules to docs**

    - Arduino Grove Color

    - Arduino Grove DLight

- Arduino Grove Ear HR

- Arduino Grove Finger HR

- Arduino Grove Haptic motor

- Arduino Grove TH02

- Pmod Color

- Pmod DLight

- Pmod Ear HR

- Pmod Finger HR

- Pmod Haptic motor

- Pmod TH02

- Added USB WiFI driver

# Python Module Index